ARTHUR RENATO MELLO

PLATAFORMA PARA DESENVOLVIMENTO E AVALIAÇÃO DE RESOLVEDORES SAT

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

2010

ARTHUR RENATO MELLO

PLATAFORMA PARA DESENVOLVIMENTO E AVALIAÇÃO DE RESOLVEDORES SAT

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

2010

SUMÁRIO

LISTA DE FIGURAS

RESUMO

Este estudo apresenta a criação de uma plataforma para o desenvolvimento e a avaliação de algoritmos que visam resolver o problema de definir a satisfatibilidade de uma fórmula em lógica proposicional. Muitos estudos já foram realizados sobre o problema da satisfatibilidade, principalmente sobre fórmulas na Forma Normal Conjuntiva. Com isso, muitas técnicas foram desenvolvidas baseadas nas características exclusivas desse formato. O algoritmo conhecido como DPLL é utilizado como base técnica para os principais resolvedores atuais. Heurísticas de aprendizado sobre erros e melhores estruturas de representação são os pontos fortes dos algoritmos mais modernos. Porém, a utilização de um formato de representação menos restritivo, não clausal, permite aos resolvedores atuarem sobre um número maior de domínios. Testes automatizados de circuitos são um bom exemplo de aplicação para um resolvedor não clausal. Dada a diversidade de aplicações, o processo de desenvolvimento de tais algoritmos exige a decisão de qual conjunto de técnicas e heurísticas deve ser utilizado para um melhor desempenho. Nesse cenário, uma plataforma de desenvolvimento robusta, que permita a implementação de estruturas e heurísticas específicas, facilita esse processo de decisão, possibilitando, assim, análises comparativas mais precisas entre diversas soluções.

ABSTRACT

This study presents a platform for the development and evaluation of algorithms designed to solve the problem of defining the satisfiability of a propositional logic formula. Many studies have been conducted on the problem of satisfiability, acting mainly on formulas in Conjunctive Normal Form. With this, many techniques were developed based on the unique characteristics of this format. The algorithm known as DPLL is mainly used as the technical basis for current solvers. Heuristics forlearning from errors and better representation structures are one of the most important characteristics of modern solvers. However, the use of a representation format less restrictive, non-clausal, allows solvers act upon a larger number of domains. Automated circuit testing is a good example of application for a non-clausal solver. Given the diversity of applications, the process of developing such algorithms requires decisions on which a set of techniques and heuristics must be used for better performance. In this scenario, a robust development platform, which allows the implementation of structures and heuristics for specific domains, eases the process of those decisions. This can make analysis between different solutions more accurate.

CAPÍTULO 1

INTRODUÇÃO

O problema de Satisfatibilidade consiste em encontrar uma valoração para as variáveis proposicionais de uma fórmula lógica proposicional, que torne a mesma verdadeira, ou seja, um modelo válido para a fórmula. Este foi o primeiro problema que se provou ser NP-Completo[9]. Apesar de amplamente investigado, o problema se mantém como foco de interesse de muitos grupos de pesquisa, dado o grande número de contextos onde a satisfatibilidade aparece como subproblema[9]. A verificação de equivalência entre circuitos combinatórios, a geração de testes automáticos para circuitos e testes de dedução lógica sobre bases de conhecimento são algumas das áreas de aplicação[4]. Desta maneira, técnicas eficientes para a solução deste problema possuem forte impacto prático em diversas áreas.

O algoritmo Davis-Putnam[5] escrito nos anos 60, foi motivado em grande parte pela busca de um provador, eficaz, de teoremas para lógica proposicional. Este foi o primeiro procedimento efetivo para a resolução do problema de satisfatibilidade que não enumerava todas as combinações da tabela verdade da fórmula. Dois anos mais tarde, o procedimento sofreu melhorias sobre as técnicas adotadas para encontrar uma solução para o problema. Esta nova versão passou a ser conhecida como Davis-Putnam-Logemann-Loveland, ou DPLL[4]. Apesar do grande avanço, o conjunto de instâncias de problemas que esse algoritmo tratava ainda era muito limitado. Nos anos seguintes, muitos resolvedores foram propostos para solucionar o problema, sendo a grande maioria baseada nos conceitos introduzidos a partir do DPLL[15, 12, 8, 6].

Uma vertente abordada por vários autores durante a década de 90, para a resolução do problema de satisfatibilidade, foi a dos algoritmos baseados em busca local. Esses algoritmos utilizam como função objetivo minimizar o número de cláusulas da fórmula que não foram resolvidas com a valoração avaliada. Um dos primeiros procedimentos que

utiliza esse princípio é o GSAT[11]. Ele se mostrou capaz de resolver uma grande variedade de problemas, considerados difíceis na época de sua apresentação, com grande eficiência. Atualmente, um dos mais eficientes procedimentos conhecidos, baseado em busca local, é o WalkSAT[10]. Apesar de eficientes, esses procedimentos não são completos, ou seja eles não podem garantir que serão capazes de encontrar uma solução caso ela exista.

Entre os algoritmos que tem como base o DPLL, as versões mais difundidas utilizam procedimentos de busca heurística, juntamente com eficientes técnicas de retrocesso sobre a árvore de decisões tomadas[12, 8, 6]. As principais distinções entre os vários métodos de resolução se referem às heurísticas empregadas na escolha de qual variável será valorada a cada interação e à cronologia do retrocesso, utilizado quando um conflito é encontrado dentro de uma valoração. A cada interação dos procedimentos, uma nova variável proposicional, presente na fórmula, tem seu valor verdade decidido e propagado. Esse processo de propagação é conhecido como Propagação de Restrições Boolenas e é uma das tarefas mais importantes de um resolvedor[8]. No que se refere a cronologia utilizada para desfazer uma escolha prévia, temos algoritmos como o próprio DPLL e o SATO[15], que sempre reconsideram a última decisão tomada. Existem também algoritmos como o GRASP[12], Chaff[8] e MiniSAT[6], que empregam elaboradas técnicas de aprendizado para escolher qual decisão necessita ser revista, independente do momento em que ela foi tomada. A capacidade de realizar o retrocesso em qualquer ordem adiciona um ganho de desempenho nos algoritmos que à utilizam.

Apesar das facilidades que as fórmulas na forma normal conjuntiva (FNC), ou forma clausal, apresentam para os principais resolvedores SAT, esse formato se mostra muito restritivo para a descrição de problemas, que seriam mais facilmente representados por fórmulas proposicionais arbitrárias. A conversão de fórmulas arbitrárias para a FNC pode gerar fórmulas com tamanho muito maior que o original, devido à necessidade de inclusão de novas variáveis proposicionais. Outro problema é a perda de informações estruturais das sentenças originais durante a conversão. Tais informações poderiam ser úteis na condução de estratégias de busca mais eficientes. Nesse cenário, a criação de procedimentos capazes de solucionar a questão da satisfatibilidade em fórmulas que não se encon-

tram na FNC, permitiria sua aplicação em um número mais abrangente de aplicações. Atualmente entre os algoritmos que focam a solução desse problema, podemos citar o NoClause[14], o PolSAT[13] e um algoritmo DPLL não clausal baseado em grafos[7].

O processo de desenvolvimento de um algoritmo resolvedor do problema de satisfatibilidade emprega um grande conjunto de decisões possíveis. Quando se trata de algoritmos que operem sobre fórmulas não clausais, muitas técnicas adotadas pelos resolvedores clausais podem ser mapeadas. A escolha desse conjunto de técnicas deve passar por um processo de análise comparativa, pois, algumas técnicas podem apresentar resultados diversos dependendo da classe de problemas tratados. Visando facilitar o desenvolvimento e a análise de algoritmos que operem com fórmulas não clausais, esse trabalho apresenta uma plataforma sobre a qual tais resolvedores podem ser implementados. Resolvedores não clausais veêm sendo o foco de muitos grupos de pesquisas[14, 13, 7] nos últimos anos, especialmente pela possibilidade de tratar um grupo menos restritivo, que as fórmulas na FNC, de problemas. A plataforma apresentada implementa as técnicas básicas do algoritmo DPLL de forma simples, permitindo, assim, que várias técnicas possam ser analisadas de forma consistente, sobre um mesmo algoritmo base. O mesmo conceito já foi apresentado para algoritmos operando sobre fórmulas clausais pelos pesquisadores responsáveis pelo MiniSAT[6], sendo este amplamanete utilizado no estudo e desenvolvimento de novas soluções.

Nos próximos capítulos do texto será explanado o estado atual dos principais algoritmos existentes para solução do problema de satisfatibilidade. No capítulo dois será realizada uma breve revisão sobre a representação lógica relevante ao problema. No capítulo seguinte descreveremos os principais algoritmos, utilizados como base para as duas principais famílias de resolvedores, os que usam a busca com retrocesso e os baseados em busca local. Também serão descritos os principais avanços das técnicas de aprendizado e suas utilizações na melhoria do processo de retrocesso para os resolvedores que atuam sobre fórmulas na FNC. No capítulo quatro veremos outras formas normais para representação de fórmulas utilizadas nos problemas de satisfatibilidade e as soluções atualmente conhecidas para sua resolução. Em seguida, a plataforma proposta será apresentada e avaliada.

Por fim, são apresentadas as conclusões e trabalho futuros.

CAPÍTULO 2

LÓGICA PROPOSICIONAL

A Lógica Simbólica, também conhecida como lógica formal, preocupa-se com a estrutura do racioncínio, não apenas em matemática mas em diversas situações cotidianas. Nesse capítulo será estudada a mais simples das lógicas simbólicas, a lógica proposicional, também conhecida como lógica booleana, uma homenagem ao lógico George Boole. Os principais conceitos, necessários para as definições dos problemas e soluções apresentados neste trabalho, serão introduzidos.

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. Por conveniência, letras maiúsculas podem ser utilizadas para representar as sentenças, definindo um *átomo* ou fórmula atômica. Os átomos também são chamados de variáveis proposicionais ou simplesmente *variáveis*. O valor "*verdadeiro*" ou "*falso*", atribuído a uma proposição qualquer, é definido como valor-verdade. Comumente, o valor verdadeiro pode ser representado pelo símbolo "V", assim como o valor falso, pelo símbolo "F".

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, utilizando os conectivos de maneira repetida, é possível a ligação de várias sentenças compostas. Dessa maneira se torna possível a representação de idéias 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 um 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.

A utilização dos parenteses serve para remover qualquer possível ambiguidade de uma fórmula lógica. Porém, $P \wedge Q$ e $(P \wedge Q)$ representam a mesma fórmula. A definição de uma ordem de precedência serve também na solução de fórmula ambiguas. A ordem de precedência adotada é: $\Leftrightarrow$, $\Rightarrow$, $\wedge$, $\vee$ e, por fim, $\neg$. Assim, as fórmulas $P \Rightarrow Q \wedge S$ e $(P \Rightarrow (Q \wedge S))$ possuem o mesmo significado.

O valor-verdade de uma fórmula lógica proposicional está relacionado ao valor-verdade de cada uma das proposições, atômicas ou compostas, que a compõem, da seguinte maneira:

- $\neg P$ será verdadeiro quando P for falso, e será falso quando P for verdadeiro. $\neg P$ é denominado negação de P.

- $(P \wedge Q)$, *conjunção* de P e Q, só será verdadeiro quando ambos forem verdadeiros e será falso em qualquer outra situação.

- $(P \vee Q)$, *disjunção* de P e Q, será verdadeiro quando ao menos um dos dois for verdadeiro e será falso quando ambos assim o forem.

- $(P \Rightarrow Q)$ será falso quando P for verdadeiro e Q for falso, caso contrário, será sempre verdadeiro. Essa fórmula é denominada "P implica Q".

- Por fim, $(P \Leftrightarrow Q)$ só será verdadeiro quando, ambos, $(P \Rightarrow Q)$ e $(Q \Rightarrow P)$ forem verdadeiros.

As relações entre os valores-verdade dos elementos que compõem uma fórmula e o valor-verdade da mesma, podem ser melhor representados por uma tabela-verdade, como a apresentada a seguir:

Tabela 2.1: Tabelas-Verdade Referentes aos Cinco Conectivos Lógicos

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
falso	falso	verdadeiro	falso	falso	verdadeiro	verdadeiro
falso	verdadeiro	verdadeiro	falso	verdadeiro	verdadeiro	falso
verdadeiro	falso	falso	falso	verdadeiro	falso	falso
verdadeiro	verdadeiro	falso	verdadeiro	verdadeiro	verdadeiro	verdadeiro

2.1 Interpretações de uma Fórmula em Lógica Proposicional

Como dito anteriormente, o valor-verdade de qualquer fórmula lógica pode ser avaliado nos termos dos valores-verdade de cada uma das proposições que a compõem. Tendo a fórmula α, representada por $(P \vee (Q \wedge \neg S))$, e supondo P e S falsos e Q verdadeiro, é possível dizer que α possui o valor verdadeiro. Essa afirmação vem do fato de a sentença $(Q \wedge \neg S)$, para um Q verdadeiro e um S falso, possuir o valor verdadeiro. Assim, a disjunção que representa a fórmula α também será verdadeira. A atribuição dos valores-verdade {F, V, F} para o conjunto de átomos $\{P, Q, S\}$, respectivamente, é denominada uma *interpretação* da fómula α. Por questões de conveniência, a interpretação anterior poderia ser representada como $\{\neg P, Q, \neg S\}$. Como cada um dos átomos que compõem α podem receber os valores-verdade V e F, mas nunca ambos, pode-se afirmar que existirão $2^3 = 8$ interpretações possíveis para a fórmula em questão.

Em linhas gerais, dada uma fórmula qualquer F composta por n atómos, que podem aparecer mais de uma vez na mesma fórmula, uma interpretação será dada pela atribuição dos valores verdadeiro e falso para cada um dos átomos da fórmula. Uma fórmula será dita verdadeira, ou válida, em uma dada interpretação, se tal atribuição lhe tornar verdadeira, ou seja, satisfaz a fórmula. Caso contrária a mesma será dita falsa para tal interpretação. Uma interpretação que satisfaça uma determinada fórmula é denominada um *modelo* para a mesma. Existirão sempre 2^n interpretações possíveis para uma fórmula lógica proposicional.

2.2 Validade e Inconsistência

Validade e inconsistência são as propriedades das fórmulas que são verdadeiras para qualquer uma de suas interpretações ou falsas para qualquer interpretação, respectivamente. Considerando a fórmula $(P \vee \neg P)$ e suas duas interpretações possíveis, $\{P\}$ e $\{\neg P\}$, o valor-verdade dela será sempre verdadeiro para qualquer uma de suas interpretações. De maneira análoga, a fórmula $(P \wedge \neg P)$ será sempre falsa para qualquer uma de suas interpretações possíveis.

Uma fórmula é dita *válida* se, e apenas se, for verdadeira para todas as suas interpretações. *Inválida* é toda fórmula que não seja válida. Por outro lado, uma fórmula é dita *inconsistente* se, e somente se, for falsa para todas as suas interpretações. Já a fórmula verdadeira para pelo menos uma de suas interpretações é dita *satisfatível* ou consistente. A negação de uma fórmula válida é inconsistente, assim como a negação de uma fórmula inconsistente é válida. Apesar de numeroso, o conjunto de interpretações possíveis para uma fórmula é sempre finito, assim sendo, examinando todas as suas interpretações é possível definir a validade ou a inconsistência de uma fórmula.

2.3 Formas Normais

A transformação de uma fórmula em outra é conseguida pela substituição da fórmula original por uma nova fórmula equivalente. Este processo se repete até que a fórmula desejada seja encontrada. Duas fórmula são ditas equivalentes quando os valores-verdade de ambas são iguais para todas as interpretações possíveis. Abaixo, algumas regras que podem ser utilizadas para transformar uma fórmula em uma nova que esteja na forma normal, por exemplo.

- $(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$ comutatividade de $\wedge$
- $(\alpha \vee \beta) \equiv (\beta \vee \alpha)$ comutatividade de $\vee$
- $((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$ associatividade de $\wedge$
- $((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$ associatividade de $\vee$

- $(\alpha \wedge (\beta \vee \gamma) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$ distributividade de $\wedge$ sobre $\vee$
- $(\alpha \vee (\beta \wedge \gamma) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$ distributividade de $\vee$ sobre $\wedge$

Um *literal* é um atómo ou a negação de um atómo, nesse caso um literal negativo. Sendo P, $\neg Q$ e S literais, a fórmula $(P \wedge \neg Q \wedge S)$ é dita como uma conjunção de literais. Por sua vez, $(P \vee \neg Q \vee S)$ é uma disjunção dos mesmos literais, ou *cláusula*. Uma fórmula está na Forma Normal Conjuntiva (FNC), quando é formada pela conjunção de disjunções de literais. Já a Forma Normal Disjuntiva (FND) é dada por uma disjunção de conjunções de literais. A fórmula representada por $((P \vee Q) \wedge (Q \vee S) \wedge T)$ é um exemplo de uma fórmula na Forma Normal Conjuntiva, visto que um único literal pode ser considerado uma disjunção. Um exemplo de uma fórmula na Forma Normal Disjuntiva pode ser representada por $((\neg Q \wedge S) \vee (P \wedge \neg T))$. Qualquer fórmula pode ser transformada em uma fórmula na forma normal pela aplicação das regras mencionadas anteriormente.

2.4 Consequência Lógica

O conceito de consequência lógica vem da necessidade de decidir se uma sentença segue logicamente de uma ou de um conjunto de sentenças. Uma fórmula α é consequência lógica de um conjunto de fórmulas se para cada interpretação onde todas as fórmulas do conjunto forem verdadeiras, a fórmula α também será. Em outras palavras dado o conjunto de fórmulas $\{F1, F2, F3\}$, α só será consequência lógica se a fórmula $((F1 \wedge F2 \wedge F3) \Rightarrow \alpha)$ for válida, ou seja, verdadeira para todas as interpretações possíveis.

Como apresentado anteriormente, se uma fórmula é válida, a negação dela deve ser inconsistente. A seguir é apresentado as regras de transformação que mostram que a fórmula $(\neg((F1 \wedge F2 \wedge F3) \Rightarrow \alpha))$ é equivalente a fórmula $(F1 \wedge F2 \wedge F3 \wedge \neg\alpha)$.

- Eliminando $\Rightarrow$, substituição de $(\alpha \Rightarrow \beta)$ por $(\neg\alpha \vee \beta)$:

  $$(\neg(\neg(F1 \wedge F2 \wedge F3) \vee \alpha))$$

- Com a aplicação da regra de equivalênica De MorganNa que define $\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$:

$$((\neg(\neg(F1 \wedge F2 \wedge F3)) \wedge \neg\alpha))$$

- Com a remoção da dupla implicação:

$$((F1 \wedge F2 \wedge F3) \wedge \neg\alpha)$$

- Com a remoção dos parenteses desnecessários:

$$(F1 \wedge F2 \wedge F3 \wedge \neg\alpha)$$

Se for encontrado um modelo para tal fórmula gerada, fica provado que a mesma não é inconsistente e, pela definição, que α não é uma consequência lógica do conjunto de fórmulas $\{F1, F2, F3\}$. Se for fixado que cada fórmula do conjunto, assim como a fórmula α, sejam disjunções de literais, a fórmula gerada estará na Forma Normal Conjuntiva. Esse fato leva a primeira grande aplicação do problema de definir a satisfatibilidade de uma fórmula na Forma Normal Conjuntiva: definir a consequência lógica, ou não, de uma sentença apartir de um conjunto de fórmulas.

Esse processo de inferência é muito utilizado por agentes lógicos. Esses agentes armazenam o conhecimento, asserções sobre o mundo atual, adquirido durante sua execução em bases de conhecimento. Essas bases podem ser compostas por fórmulas lógicas proposicionais e o processo de inferência lógica pode ser utilizado, recaindo, assim, em um problema de definir a satisfatibilidade de uma fórmula. É sobre as principais técnicas conhecidas para a solução desse problema que os próximos capítulos deste trabalho irão tratar.

CAPÍTULO 3

SATISFATIBILIDADE EM FNC

Os principais algoritmos conhecidos para a resolução do problema de satisfatibilidade em fórmulas lógicas proposicionais se dividem em duas principais famílias: uma baseada em busca com retrocesso e outra baseada em busca local. A expressão busca com retrocesso é utilizada para indicar os algoritmos, que durante a busca de um modelo para a fórmula lógica proposicional, definem o valor-verdade para uma proposição de cada vez e efetuam o retrocesso caso alcançem uma interpretação que não satisfaça a fórmula. O espaço de busca do problema de encontrar um modelo para uma fórmula lógica proposicional, compreende todas as interpretações possíveis para tal fórmula. Como apresentado anteriormente, existem 2^n interpretações possíveis para uma fórmula com n váriaveis proposicionais. Os algoritmos baseados em busca com retrocesso realizam uma exploração sistemática deste espaço de busca. Esse caráter sistemático é alcançado mantendo-se todas as decisões de qual valor-verdade foi atribuído a uma váriavel proposicional e quais ainda não foram decididas. Durante este processo, o conjunto de valores-verdade atribuídos a apenas uma parte das variáveis proposicionais da fórmula é denominado valoração parcial.

Já para a família de algoritmos baseados em busca local, as valorações parciais do processo para encontrar um modelo não são importantes. Esses algoritmos operam sobre uma única interpretação a cada momento, com todas as váriaveis proposicionais possuindo um valor-verdade atribuído. Caso essa interpretação não satisfaça a fórmula em questão, o algoritmo abandonará a atual e passará a operar sobre uma interpretação vizinha dentro do espaço de busca. Uma interpretação é definida como vizinha de outra dentro de um espaço de busca quando o conjunto de valores atribuídos difere em apenas uma única váriavel proposicional.

Apesar dos bons resultados alcançados com ambas as abordagens, os procedimentos baseados em buscas com retrocesso costumam receber maior atenção por parte dos pes-

quisadores, inicialmente pelo seu melhor desempenho, mas principalmente pelo fato de serem algoritmos de busca completos, ao contrário dos algoritmos baseados em busca local. O fato de não realizarem uma busca sistemática dentro do espaço de busca, não garante que todo o espaço será percorrido. Dessa maneira, o fato de um algoritmo de busca local não ser capaz de encontrar um modelo para uma fórmula, não garante que tal modelo não exista.

Nas seções que seguem apresentaremos os principais algoritmos resolvedores do problema de satisfatibilidade para fórmulas lógicas e aprofundaremos um pouco mais a discussão sobre as principais características de cada uma das duas famílias de busca. Iniciamos com o algoritmo DPLL, utilizado como base da grande maioria de algoritmos baseados em busca com retrocesso[15, 12, 8, 6].

3.1 O Algoritmo DPLL

O algoritmo conhecido como DPLL se refere ao procedimento denominado Davis-Putnam-Logemann-Loveland, introduzido em 1962 pelos pesquisadores Martin Davis, George Logemann e Donald W. Loveland[4]. Essa pesquisa introduziu melhorias ao trabalho apresentado dois anos antes por Martin Davis e Hilary Putnam[5]. No ínicio da década de 20 havia um grande esforço em pesquisas visando a prova de teoremas matemáticos através da aplicação de métodos matemáticos sobre fórmulas lógicas. Nesse contexto, o problema de determinar a satisfatibilidade de uma fórmula lógica também se mostrava relevante. Vários provadores foram sugeridos, sempre baseados em buscas dentro do conjunto de todas as interpretações da fórmula. Apesar de encontrarem soluções para alguns casos, normalmente terminavam executando seus procedimentos indefinidamente. O procedimento Davis-Putnamm foi introduzido nesse cenário, sendo capaz de tratar fórmulas mais complexas que os demais resolvedores da época.

O procedimento DPLL opera sobre fórmulas lógicas proposicionais na forma normal conjuntiva, ou FNC. Dessa maneira, caso a fórmula inicial não se encontre na FNC, uma conversão é necessária. Como demonstrado no capítulo anterior, qualquer fórmula lógica proposicional pode gerar uma fórmula equivalente na FNC, apenas com a aplicação de

regras de equivalência. A adoção de fórmulas nessa forma normal, permite ao algoritmo definir se uma valoração parcial do conjunto de variáveis que compõem o problema é um modelo da fórmula.

Se algum literal de uma cláusula da fórmula for verdadeiro para uma dada valoração, parcial ou não, toda a cláusula pode ser considerada verdadeira para esta valoração. Isto ocorre pelo fato de cada cláusula ser uma disjunção de literais e, dentro da semântica definida, para uma disjunção ser verdadeira apenas um dos seus disjuntos necessita ser.

Encerrado o processo de transformação da fórmula, o algoritmo utilizará duas regras que buscam encontrar uma valoração que a deixe verdadeira. Essas regras são conhecidas como regra do literal puro e a regra da cláusula unitária.

Um literal puro é definido como um literal que só aparece em uma forma, positivo ou negativo, em toda a fórmula. A cada decisão tomada pelo procedimento, sobre o valor-verdade de uma váriavel proposicional, deve ser escolhida aquela capaz de maximizar o número de cláusulas decididas como verdadeiras e minimizar o número das decididas como falsas. Isso decorre do fato do procedimento buscar um modelo para a fórmula. Logo, a variável proposicional do literal puro deve ser valorada de tal forma que todas as ocorrências do literal sejam verdadeiras. Desta maneira, todas as cláusulas em que ele está presente serão consideradas verdadeiras. Tal regra nunca definirá uma cláusula como falsa, o que assegura que a valoração atribuída a variável não necessitará ser revista em momento algum.

No contexto do algoritmo DPLL, um literal pode ser considerado puro mesmo que se apresentem as duas formas, positivo e negativo, na mesma fórmula. Toda cláusula considerada verdadeira para a valoração atual, pode ser desconsiderada do restante da fórmula. Porém, essa remoção só é relevante para a valoração atual, caso essa seja revista e a cláusula deixe de ser verdadeira, ela deve voltar a ser considerada. Assim, se um literal é puro no subconjunto de cláusulas que ainda não foram decididas como verdadeira, a regra do literal puro pode ser aplicada. Mas nesse caso, a decisão pode necessitar ser revista no futuro.

Uma cláusula é considerada unitária quando não possui nenhum literal verdadeiro e

apenas um literal que ainda não possui um valor-verdade atribuído. Sempre que uma variável proposicional é atribuída, seus literais decididos como falsos se tornam irrelevantes para as cláusulas em que estão presentes, visto que eles não auxiliam na busca de um modelo. Isso ocorre de maneira análoga a decisão de uma cláusula como verdadeira. Assim, esse literal passa a não figurar para o algoritmo, na contagem dos literais relevantes da cláusula. Caso essa contagem chegue a zero, a cláusula pode ser considerada falsa para a valoração atual que, por sua vez, não é um modelo da fórmula. Caso a contagem aponte um único literal, uma nova cláusula unitária foi encontrada. A variável proposicional representada por esse literal único deverá receber um valor-verdade que torne o literal verdadeiro, evitando que a cláusula seja considerada falsa. A aplicação da regra da cláusula unitária pode levar ao surgimento de novas cláusulas unitárias, pois a proposição decidida pode possuir literais complementares em outras cláusulas da fórmula. Esse efeito de atribuição cascateado é uma das principais características do algoritmo DPLL e é conhecido como Propagação de Restrições Unitárias, ou, simplesmente, propagação unitária.

Após todas as aplicação possíveis das regras de literal puro e cláusula unitária serem executadas, caso um modelo não tenha sido encontrado, o procedimento necessita tratar as cláusulas da fórmula que ainda não foram decididas. Porém, nenhuma variável proposicional, com literais positivos e negativos presentes em cláusulas ainda não deicididas da fórmula, pode ser decidida de forma trivial. O algoritmo deve aplicar uma regra conhecida como ramificação. Na regra de ramificação, uma variável qualquer, da lista das que ainda não possuem valor-verdade definido, deve ser escolhida e definida arbitrariamente. A aplicação da regra de ramificação pode permitir o surgimento de novas cláusulas unitárias ou literais puros, permitindo assim que as regras respectivas sejam aplicadas novamente. A decisão do valor por essa regra pode, em passos futuros do algoritmo DPLL, se mostrar equivocada para a busca de um modelo. O valor-verdade atribuído pode não fazer parte de nenhum modelo da fórmula, porém sua decisão não define nenhuma cláusula da fórmula como falsa. Caso isso ocorra, o algoritmo deverá retroceder até o ponto onde a decisão foi tomada e alterá-la, atribuindo o valor oposto à variável em questão. O controle das variáveis que tiveram seus valores atribuídos desta maneira pode ser representado

como uma árvore de decisão. Quando todas as alternativas de valoração se esgotarem, o algoritmo DPLL poderá afirmar que a fórmula em questão não é satisfatível.

O algoritmo DPLL pode ser representado pelo pseudo código apresentado no algoritmo 3.1, sendo F uma fórmula na FNC e VP uma valoração parcial para esta fórmula.

Algoritmo 3.1: Algoritmo DPLL

```
1  DPLL(F, VP)
2      se todas as cláusulas de F forem verdadeiras
3          então retorne verdadeiro;
4      se alguma cláusula de F for falsa
5          então retorne falso;
6
7      se ocorrer um literal puro L em F
8          então retorne DPLL(DECIDE(L, F), ATRIBUI(L, VP));
9      se ocorrer uma cláusula unitária C em F
10         então retorne DPLL(DECIDE(C, F), ATRIBUI(C, VP));
11
12     L := ESCOLHA_LITERAL();
13     retorne DPLL(DECIDE(L, F), ATRIBUI(L, VP))
14          ou DPLL(DECIDE(NÃO(L), F), ATRIBUI(NÃO(L), VP));
```

Para exemplificar as etapas do processo, a seguir demonstramos todos os passos executados pelo algoritmo DPLL para decidir se a sentença:

$$(P \vee Q) \wedge (S \vee \neg T) \wedge (R) \wedge (\neg S \vee \neg T) \wedge (\neg Q \vee \neg S) \wedge (R \vee \neg U) \wedge (\neg S \vee T) \wedge (\neg R \vee U)$$

é ou não satisfatível.

- Inicialmente o algoritmo executa, se possível, a regra do literal puro. Na fórmula em questão o literal P é puro, pois a fórmula não possui nenhuma ocorrência de $\neg P$. Visando tornar o literal P verdadeiro, o algoritmo atribui o valor-verdade Verdadeiro para a proposição. Resultando na fórmula:

  Interpretação {P = {V}}

  $$(\{V\} \vee Q) \wedge (S \vee \neg T) \wedge (R) \wedge (\neg S \vee \neg T) \wedge (\neg Q \vee \neg S) \wedge (R \vee \neg U) \wedge (\neg S \vee T) \wedge (\neg R \vee U)$$

- Toda disjunção que possui um disjunto Verdadeiro é, ela própria, verdadeira. E desse modo, pode ser removida da fórmula. A remoção da cláusula $(\{V\} \vee Q)$ torna o literal $\neg Q$ puro. Pela aplicação da regra, a proposição Q deve receber o valor-verdade Falso:

  Interpretação {P = {V}, Q = {F}}

  $(S \vee \neg T) \wedge (R) \wedge (\neg S \vee \neg T) \wedge (\{V\} \vee \neg S) \wedge (R \vee \neg U) \wedge (\neg S \vee T) \wedge (\neg R \vee U)$

- As cláusulas verdadeiras são novamente removidas. Não há mais nenhum literal puro presente na fórmula. O algoritmo busca agora as cláusulas unitárias, como (R). Para tornar a cláusula verdadeira, o valor-verdade Verdadeiro deve ser atribuído a proposição R:

  Interpretação {P = {V}, Q = {F}, R = {V}}

  $(S \vee \neg T) \wedge (\{V\}) \wedge (\neg S \vee \neg T) \wedge (\{V\} \vee \neg U) \wedge (\neg S \vee T) \wedge (\{F\} \vee U)$

- Um disjunto Falso é irrelevante para uma disjunção. Assim sendo, a cláusula $(\{F\} \vee U)$ pode ser considerada uma cláusula unitária. Como tal, a proposição U deve ter o valor verdadeiro na interpretação testada:

  Interpretação {P = {V}, Q = {F}, R = {V}, U = {V}}

  $(S \vee \neg T) \wedge (\neg S \vee \neg T) \wedge (\neg S \vee T) \wedge (\{F\} \vee \{V\})$

- Nesse ponto o algoritmo é incapaz de aplicar as regras do literal puro ou da cláusula unitária. O algoritmo então, utilizando a regra da ramificação, escolhe atribuir o valor verdadeiro à proposição S. Essa escolha é arbitrária, e como tal, deve ser assinalada para um possível retrocesso:

  Interpretação {P = {V}, Q = {F}, R = {V}, U = {V}, S* = {V}}

  $(\{V\} \vee \neg T) \wedge (\{F\} \vee \neg T) \wedge (\{F\} \vee T)$

- Novamente o algoritmo precisa da regra de ramificação para tomar a próxima decisão. O algoritmo atribui o valor verdadeiro para a proposição T:

Interpretação {P = {V}, Q = {F}, R = {V}, U = {V}, S* = {V}, T* = {V}}

$(\{F\}) \wedge (\{V\})$

- A última atribuição gerou uma cláusula vazia, consequentemente, falsa. Como a interpretação atual não é um modelo para a fórmula, o processo de retrocesso deve ser realizado. Inicialmente, a última decisão será desfeita e o valor Falso será atribuído à proposição T. Porém, essa decisão também irá gerar uma cláusula vazia. Assim sendo, o algoritmo deve retroceder até o instante anterior a atribuição de S como Verdadeiro e tentar, agora, atribuir o valor-verdade Falso a S:

  Interpretação {P = {V}, Q = {F}, R = {V}, U = {V}, S* = {F}}

  $(\{F\} \vee \neg T) \wedge (\{V\} \vee \neg T) \wedge (\{V\} \vee T)$

- Dessa vez a valoração parcial testada gerou uma única cláusula. Essa cláusula é unitária no literal $\neg T$ que, por consequência, também é um literal puro. Atribuindo o valor-verdade Falso para a proposição T, o algoritmo é capaz de definir que a fórmula em questão é satisfatível. O algoritmo pode fazer tal afirmação, visto que o seguinte modelo para a fórmula foi encontrado:

  Interpretação {P = {V}, Q = {F}, R = {V}, U = {V}, S* = {F}, T* = {F}}

3.2 Melhorias introduzidas pelo Algoritmo SATO

Durante a década de 90, o interesse sobre os problemas de satisfatibilidade foi renovado com a propagação de novos resolvedores que utilizavam e melhoravam as regras introduzidas com o procedimento DPLL. Dentre esses resolvedores, os pesquisadores Hantao Zhang e Mark E. Stickel apresentaram o algoritmo SATO[15], um acrônimo das iniciais em inglês de "Teste Otimizado de Satisfatibilidade". Os pesquisadores introduziram melhorias que visavam aperfeiçoar tanto a regra de ramificação, quanto introduzir análise de conflito ao DPLL. Um conflito ocorre sempre que uma decisão acaba gerando uma cláusula vazia e, por definição, não faz parte de um modelo da fórmula. Com a aplicação dessas

alterações, o resolvedor SATO se mostrou capaz de tratar de maneira mais eficiente que seus antecessores alguns conjuntos de fórmulas lógicas proposicionais.

Os pesquisadores Crawford e Auton[3] introduziram um procedimento capaz de realizar o processo de encontrar uma cláusula unitária e propagar a decisão da nova proposição decidida em tempo linear. Como demonstrado no exemplo de execução do algoritmo DPLL na seção anterior, a cada passo o algoritmo necessita percorrer a fórmula seja para verificar o surgimento de cláusulas especiais, seja para buscar possíveis literais unitários ou ainda, para atribuir os novos valores-verdade decididos a cada literal. Se tal processo é simples para uma fórmula do tamanho da que foi apresentada no exemplo, para fórmulas com centenas de cláusulas e cada cláusula com dezenas de literais, realizar essa busca com frequência pode ter um forte impacto no desempenho do algoritmo.

Segundo a implementação de Crawford e Auton[3], para cada variável proposicional presente na fórmula lógica, duas listas devem ser criadas e atualizadas a cada interação do processo: uma com todas as cláusulas onde o literal representante da proposição aparece na forma negada, e outra com as cláusulas onde ele aparece na forma positiva. Já para as cláusulas, um contador é atribuído mostrando o número de literais, presentes na cláusula, que ainda não foram valorados. Caso a cláusula já tenha sido solucionada, um valor inválido pode ser armazenado nesse contador. A cada alteração na valoração de uma variável da fórmula, os contadores das cláusulas onde tal variável possui literais devem ser atualizados. Caso a variável seja valorada como verdadeira, todas as cláusulas onde seu literal negado aparece devem decrementar os seus contadores. Caso ela seja valorada como falso, as cláusulas que possuem seu literal na forma positiva devem decrementar seus contadores. Quando o contador de uma cláusula tem seu valor igual a um, essa cláusula é considerada uma cláusula unitária. Caso o contador seja zerado, essa cláusula é dita vazia e a valoração atual não representará um modelo para a fórmula. Assim, a cláusula não necessita ser percorrida, apenas seus contadores são manipulados. Baseado nessa proposta, os pesquisadores do algoritmo SATO aplicaram alterações que reduzem, ainda mais, o tempo para a propagação unitária.

No procedimento adotado pelo SATO para a propagação unitária, cada cláusula é

representada como uma lista de literais e dois ponteiros devem ser armazenados. Um para o primeiro literal presente na cláusula, denominado cabeça e outro para o último, denominado cauda. Para cada proposição da fórmula, quatro listas devem ser mantidas:

- Uma com todas as cláusulas onde o literal que a representa aparece na forma positiva, como cabeça de uma cláusula.
- Outra com as cláusulas que também possuem literais positivos, mas com esses aparecendo na cauda.
- A terceira com as cláusulas onde os literais aparecem na forma negativa como cabeça.
- A última com as cláusulas onde os literais negativos aparecem na cauda.

Quando o algoritmo define uma variável como verdadeira, todas as cláusulas das duas listas onde seus literais aparecem na forma positiva passam a ser ignoradas, tanto na cabeça quanto na cauda. Essas cláusulas são consideradas resolvidas, pois um de seus disjuntos foi valorado como verdadeiro. Elas devem, por isso, ser removidas das respectivas listas das proposições.

Já para as cláusulas onde os literais aparecem na forma negativa, na cabeça ou na cauda, desloca-se o ponteiro de cabeça ou de cauda da própria cláusula, para um literal vizinho. No caso do literal negado da váriavel estar presente na cabeça da cláusula, o ponteiro de cabeça é deslocado para o próximo literal. Mas no caso do literal negado estar presente na cauda, o ponteiro de cauda é movido para o literal anterior. Como os literais da proposição valorada não fazem mais parte da cabeça e nem da cauda dessas cláusulas, as mesmas devem ser removidas das listas de cláusulas das variáveis.

O processo de deslocamento do ponteiro de cabeça ou cauda da cláusula ocorre, pois o literal para o qual ele apontava é considerado falso na interpretação atual, por isso, irrelevante para a cláusula. O papel desses ponteiros é monitorar alguns literais da cláusula, para evitar que a cláusula inteira necessite ser pecorrida a cada decisão que envolva um de seus literais. Se dois literais da cláusula, no caso a cabeça e a cauda, ainda não foram decididos, nenhuma regra especial pode ser aplicada sobre ela. Porém, se não for possível

encontrar dois literais não valorados, ou uma cláusula unitária ou uma cláusula vazia estará presente na fórmula. No caso de ser uma cláusula unitária, os ponteiros de cabeça e cauda estarão apontando para o mesmo literal, e esse literal representa a proposição que deve ser valorada na regra da cláusula unitária. Se os ponteiros de cabeça e cauda não apontam para nenhum literal, a cláusula é vazia e a valoração atual não representa um modelo da fórmula.

Dessa maneira, o processo de troca dos ponteiros de cabeça e cauda deve sempre buscar um literal que não tenha sido valorado como falso. Caso um literal não decidido seja encontrado, ele se torna o novo destino do ponteiro de cabeça ou cauda, e as listas da proposição que representa devem ser atualizadas com a nova cláusula. Se por outro lado, um literal decidido como verdadeiro for encontrado, a cláusula passa a ser ignorada e por isso, deve ser removida das respectivas listas das proposições.

Todo o procedimento funciona de maneira análoga quando atribuímos o valor falso a uma proposição, bastando que apenas as cláusulas presentes nas respectivas listas da proposição sejam visitadas e alteradas conforme a necessidade. Esta é a principal vantagem da utilização dessas listas: facilitar o acesso as cláusulas relevantes a cada nova atribuição. Porém, a manutenção dessas listas, com as respectivas remoções e inserções de cláusulas, aumenta o tempo de processamento necessário para o algoritmo encontrar uma solução. Outro ponto a ser considerado, é o fato de que como apenas a cabeça e a cauda de cada cláusula são levados em consideração, é possível que uma valoração parcial já seja um modelo e o algorimo, ainda assim, não encerre o seu processamento. Para isso, basta que uma proposição decidida torne verdadeira todas as cláusulas de uma fórmula, porém os literais dessa proposição não façam parte nem da cabeça nem da cauda de cada cláusula.

Durante a execução do procedimento descrito pelo algoritmo DPLL, após a execução repetida da remoção de literais puros e da propagação unitária, o processo chega em um ponto onde é necessário decidir qual variável será utilizada na regra da ramificação e qual valor-verdade deve ser atríbuido. Usualmente essa decisão é arbitrária, porém diferentes heurísticas podem ser utilizadas para melhorar o desempenho do resolvedor.

A heurística utilizada depende do tipo de problema tratado. Os criadores do algoritmo SATO utilizaram várias heurísticas diferentes para realizar essa escolha. Uma delas define que deve ser escolhido um literal presente na menor cláusula positiva, ou seja, uma cláusula que não apresente nenhum literal na forma negativa, e realizar a ramificação sobre ele. Já outra, diz que o literal escolhido deve fazer parte de uma cláusula binária, com apenas dois literais facilitando dessa forma, o surgimento de novas cláusulas unitárias nessa decisão ou em um possível retrocesso. Uma terceira heurística define que, sendo n o número de cláusulas onde o literal escolhido aparece na forma negativa e p o número de cláusulas onde ele aparece na forma positiva, o literal escolhido deve maximizar a fórmula $(n+1) * (p+1)$[3].

Visando chegar em uma solução que combinasse várias heurísticas, o algoritmo SATO apresentou um novo procedimento. Dentro do conjunto de todas as cláusulas ainda não definidas, forma-se um conjunto composto pelas cláusulas com o menor número de literais, para dele escolhermos algumas cláusulas. A quantidade de cláusulas escolhidas será proporcional à quantidade total de cláusulas da fórmula. Entre todos os literais das cláusulas escolhidas, escolheremos aquele que possui o maior valor da função descrita anteriormente. Dessa maneira a ramificação se adapta à fórmula a ser resolvida.

Sempre que uma cláusula vazia é encontrada, onde todos os seus literais foram valorados para falso, o algoritmo necessita realizar um retrocesso na valoração atual, visto que a mesma não representa uma interpretação capaz de tornar a fórmula verdadeira. Realizar esse retrocesso de maneira inteligente permitirá ao resolvedor não aplicar a regra de ramificação sobre variáveis desnecessárias, reduzindo assim o espaço de busca do problema. A heurística básica de retrocesso apresentada no algoritmo SATO define que, quando uma cláusula vazia é encontrada, deve-se definir o conjunto de todos os literais cuja valoração foi decisiva para que a cláusula em questão se tornasse vazia. Se o último literal valorado pela regra de ramificação não estiver presente nesse conjunto, não há a necessidade de testarmos o seu segundo valor-verdade, sendo assim, devemos retroceder até uma decisão anterior. Essa ideia, associada a manutenção das valorações parciais que geraram cláusulas vazias, a fim de evitar a repetição de erros, são as ideias básicas por

trás de uma heurística de retrocesso muito mais robusta apresentada na próxima seção.

A seguir é apresentado um exemplo de monitoramento e substituição dos literais cabeça e cauda de cada cláusula:

- Fórmula inicial, composta de duas cláusulas: $(\neg B \vee \neg C) \wedge (\neg A \vee \neg B \vee \neg C)$.
- A lista de literais posicionados na cabeça de cada cláusula é composta de: $\neg B$ na primeira cláusula e $\neg A$ na segunda.
- A lista de literais posicionados na cauda de cada cláusula é composta apenas por: $\neg C$ na primeira cláusula e $\neg C$ na segunda.
- Atribuindo B como verdadeiro: $(F \vee \neg C) \wedge (\neg A \vee F \vee \neg C)$.
- Como a atribuição torna um literal cabeça como falso, ele deve ser substituído. Dessa maneira a lista de literais posicionados na cebeça será: $\neg C$ na primeira cláusula e $\neg A$ na segunda.
- Atribuindo A como verdadeiro: $(F \vee \neg C) \wedge (F \vee F \vee \neg C)$.
- Novamente um literal cabeça torna-se falso. Com a substituição a lista de literais cabeça será: $\neg C$ na primeira cláusula e $\neg C$ na segunda.
- Como o mesmo literal é apontado como cabeça e cauda de cada uma das cláusulas, as duas são consideradas cláusulas unitárias: $(\neg C) \wedge (\neg C)$

3.3 O Algoritmo Grasp: Técnicas de Aprendizado de Novas Cláusulas

Algoritmo de Busca Genérica para o Problema de Satisfatibilidade, ou na sigla em inglês, Grasp[12], é um resolvedor baseado no algoritmo DPLL que tem por objetivo unificar várias técnicas de busca utilizadas anteriormente. Proposto por João P. Marques Silva e Karem A. Sakallah, se baseia no fato de que a aplicação repetida da regra de ramificação acabará, cedo ou tarde, gerando uma cláusula vazia. Quando isso ocorre,

um conflito foi gerado. Através de uma análise criteriosa dos conflitos encontrados, o algoritmo é capaz de definir as suas causas, ou seja, as atribuições de valores-verdade equivocadas que acabaram por levar ao problema. Com essa informação, é possível a realização do retrocesso na primeira decisão equivocada responsável pelo conflito atual, e não mais a revisão da última decisão tomada. Esse sistema é definido como retrocesso não cronológico.

O Grasp é capaz de desfazer e, em seguida, testar o valor complementar, de qualquer decisão prévia, não sendo necessário, assim, testar sempre o valor complementar das últimas decisões tomadas, sequencialmente. Realizando esses retrocessos não sequenciais o algoritmo é capaz de uma redução significativa do espaço de busca visitado pelo algoritmo, reduzindo o tempo necessário para a resolução do mesmo. Dentre outras melhorias introduzidas, a manutenção de valorações parciais que geraram conflitos anteriormente é uma das principais. Com elas o algoritmo é capaz de evitar que os mesmos equívocos se repitam nas decisões futuras.

O processo de busca realizado pelo Grasp segue os mesmos passos introduzidos pelo algoritmo DPLL, no que se refere ao processo de valoração dos literais e a expansão de suas implicações. Inicialmente ocorrem as aplicações repetidas da regra do literal puro e da cláusula unitária. Após isso, uma variável deve ser escolhida e seu valor-verdade decidido. Esse processo se repete enquanto houverem cláusulas com valor-verdade não definido e nenhuma cláusula vazia for encontrada, sempre repetindo as aplicações das regras anteriores quando possível.

Caso a execução desses passos gere uma cláusula vazia na fórmula, o procedimento de análise de conflito precisa ser executado para definir quais decisões foram as responsáveis. A partir dessa informação, é necessário garantir que o conjunto de atribuições que geraram o conflito não ocorra novamente. Para isso, uma nova cláusula é criada e adicionada à fórmula lógica proposicional atual, para que a informação seja considerada a cada nova decisão tomada. Essa cláusula é formada por literais que representam as proposições reponsáveis pelo conflito, de maneira que caso as mesmas decisões sejam tomadas, essa cláusula se torne vazia. Para possibilitar a análise e o aprendizado através de conflitos o

algoritmo utiliza algumas estruturas diferenciadas, sendo a principal o grafo de implicação.

No grafo de implicação, um grafo direcionado, cada vértice representa uma variável que possui um valor-verdade atribuído na valoração atual. As arestas presentes no grafo demonstram as relações de dependência entre as atribuições. Cada aresta direcionada define que o vértice de origem está presente na cláusula unitária que resultou na implicação do valor descrito no vértice de destino. Ou seja, se uma decisão é tomada a partir de uma cláusula unitária que não estava presente na fórmula inicial, uma ou mais atribuições foram necessárias para a geração dessa cláusula. Essas atribuições, necessárias para a geração de uma cláusula unitária, são todas aquelas que tornaram todos os literais da cláusula, menos um, falsos na valoração atual. A mesma informação pode ser armazenada para atribuições provenientes de aplicações da regra do literal puro. Sempre que um novo literal puro surge na fórmula, algum conjunto de atribuições foi necessário para a remoção de todos os seus literais complementares.

Desta forma, a qualquer momento, é possível especificar o conjunto de atribuições responsável por uma decisão, apenas obtendo o conjunto de todos os vértices que possuem arestas chegando ao vértice que representa a decisão. Vértices que não possuem arestas incidentes obtiveram seus valores não pela implicação, mas através da regra de ramificação. Para cada vértice do grafo será definido um valor referente ao seu nível de decisão. Esse nível descreve quando, entre todas as decisões tomadas, aquela ocorreu.

A primeira valoração de uma fórmula, representada por um vértice no grafo de implicação, possui nível de decisão igual a zero. Cada uma das decisões subsequentes, que não ocorrerem através de implicações, terão nível de decisão incrementado sequencialmente. Já para as valorações decorrentes de uma implicação, a aplicação das regras de literal puro e de cláusula unitária, terão nível de decisão igual ao maior dentre os níveis de decisão das atribuições de variáveis que compõem o conjunto responsável pela geração da implicação.

Na figura 3.3 será apresentado um grafo de implicação com três decisões iniciais, A, B e C, que implicaram em uma quarta decisão D. Os níveis de decisão são apresentados dentro de conchetes.

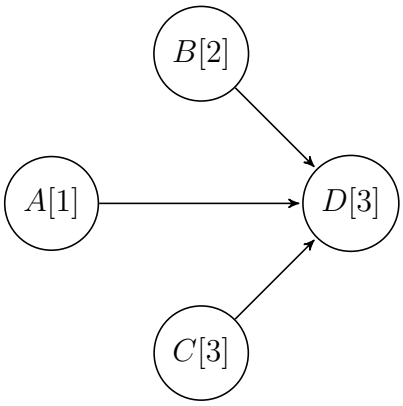

Figura 3.1: Grafo de Implicação com três decisões iniciais

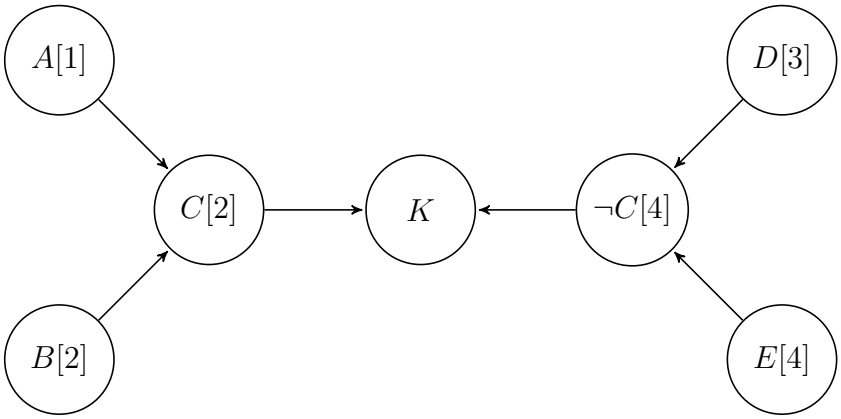

Figura 3.2: Grafo de Implicação contendo um conflito

Quando uma cláusula vazia é encontrada, um novo vértice será adicionado e as arestas que incidirão sobre ele serão provenientes dos vértices que representam as decisões das variáveis que resultaram na cláusula vazia. Esse vértice será responsável por representar o conflito encontrado.

Na figura 3.3, um grafo de implicação contendo um conflito, representado pelo nodo K. As decisões iniciais, A e B, implicaram na decisão C, em contrapartida, as decisões D e E implicaram na decisão não-C, ou seja, o complemento da decisão C. Como uma váriavel lógica não pode ser verdadeira e falsa ao mesmo tempo, duas decisões complementares não podem existir em uma solução válida para o problema de satisfatibilidade. O nodo representante do conflito será ligado, por arestas, aos nodos que reprensentam as decisões complementares.

Partindo do vértice que representa o conflito e buscando o conjunto de variáveis que é suficiente para gerá-lo, o procedimento de análise deve formar um grupo com todos os vértices antecessores. Ou seja, todos aqueles que possuem arestas incidindo no conflito. Esse conjunto nunca deve ser vazio, pois caso isso ocorra a última decisão tomada gerou automaticamente um conflito. O retrocesso deve ser realizado diretamente sobre ela, e o processo de decisão sobre qual valor-verdade deve ser atribuído a uma proposição deve ser revisto.

Caso alguma atribuição presente no conjunto responsável pelo conflito seja proveniente de uma implicação, ela será substituída pelas atribuições que geraram tal implicação. Sendo assim, seu vértice deve ser removido do conjunto e aqueles responsáveis por tal implicação devem ser colocados no lugar. Esse procedimento de substituição deve ser repetido até que no conjunto restem apenas vértices que não foram gerados a partir de implicações. Cada variável representada por um vértice do conjunto deve ser adicionada a uma nova cláusula, que representará todo o conhecimento adquirido com este conflito.

Se na valoração atual a variável possuir o valor verdadeiro, seu literal negado deve ser inserido na cláusula aprendida. Caso contrário, se o valor atribuído a variável for falso, incluiremos na cláusula o literal positivo. A cláusula aprendida passa a ser considerada como uma cláusula qualquer da fórmula, devendo ser considerada a cada modificação na valoração corrente. Se uma cláusula aprendida se torna vazia, a valoração atual possui os mesmos erros que levaram ao conflito originário da cláusula.

Com a valoração atual da fórmula lógica, a nova cláusula inserida será uma cláusula vazia, o que não é interessante para a resolução do problema de satisfatibilidade. Logo, o resolvedor deve desfazer alguma das decisões anteriores. Para tanto, o Grasp usa como heurística desfazer a decisão sobre a variável presente na cláusula aprendida, que possui o maior nível de decisão no grafo atual. Em um primeiro momento essa variável será a última valorada através da regra de ramificação. Assim que a valoração for desfeita, seu vértice deve ser removido do grafo, assim como todos os vértices sucessores. O procedimento é análogo, caso o maior nível de decisão encontrado na cláusula aprendida a partir do conflito seja menor que o nível de decisão atual. Porém, nesse caso, além de remover o

vértice da variável escolhida e seus sucessores, o algoritmo deve remover todo vértice que possua nível de decisão maior que o da variável em questão.

A cláusula introduzida pelo conflito passa a ser uma cláusula unitária e a decisão implícita será a do último literal decidido pela regra de ramificação, com seu valor complementar. A inserção dessa decisão no grafo segue o mesmo procedimento descrito, porém dessa vez, a decisão não é mais uma escolha da regra de ramificação, mas o resultado de uma implicação gerada por uma cláusula unitária, a cláusula aprendida. Sendo assim, o vértice possuirá arestas incidentes e no caso de um novo conflito, não será adicionado na cláusula aprendida.

A seguir será apresentada uma sequência de decisões e implicações tomadas sobre a fórmula lógica proposicional composta de uma única cláusula e o grafo de implicação resultante:

$$(P \vee \neg Q \vee R \vee S)$$

- Inicialmente atribuimos a proposição P com o valor-verdade falso. Por ser a primeira atribuição resultante da regra de ramificação, seu nível de decisão será igual a 1.

- Em seguida, atribuimos a proposição Q com o valor-verdade verdadeiro. Seu nível de decisão será igual a 2.

- Por fim, atribuimos a proposição R com o valor-verdade Falso. Seu nível de decisão é 3.

- Dado o estado atual da fórmula, a implicação da proposição R como verdadeira é possível. Seu nível de decisão será igual ao maior nível das decisões responsáveis por sua implicação. Logo, seu nível de decisão será igual a 3. A figura 3.3 mostra o grafo de implicação correspondente.

O processo de geração de uma cláusula aprendida apartir do conflito representado pelo nodo K no grafo de implicação da figura 3.3. Esse conflito foi gerado pelas decisões da váriavel P como falso e da váriavel Q como verdadeiro. Essas decisões implicaram em S verdadeiro e em R verdadeiro, que por sua vez implicou o valor falso na váriavel S. Como

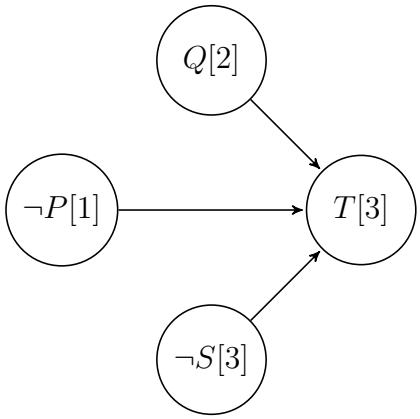

Figura 3.3: Grafo de Implicação com níveis de decisão

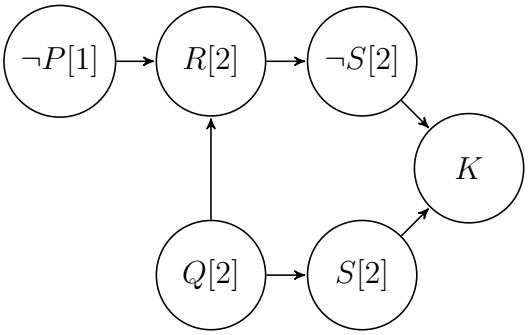

Figura 3.4: Processo de Geração de Cláusula Aprendida

a váriavel proposicional S não pode ser verdadeira e falsa ao mesmo tempo, um conflito foi encontrado.

- O conjunto inicial de decisões responsáveis pelo conflito K será formado pelos vértices que representam as decisões S e $\neg S$.
- Ambos os nodos foram produtos de uma inferência, logo devemos substituí-los pelos vértices responsáveis pela inferência, ficando assim com um conjunto contento os vértices que representam as decisões R e Q.
- Como a decisão da proposição R também é decorrente de uma inferência, ele será substituído pelo seu responsável. O conjunto final de decisões responsáveis pelo conflito K será $\neg P$ e Q.
- Esse conjunto gera a cláusula $(P \vee \neg Q)$ que deve ser adicionada a fórmula original. Como na interpretação atual essa cláusula é vazia, devemos revisar a decisão de

alguns de seus disjuntos, aqueles com os maiores valores para o nível de decisão. Nesse caso, devemos revisar a decisão que atribuiu o valor-verdade verdadeiro para a proposição Q.

Apesar das melhorias significativas introduzidas por este procedimento de análise de conflitos, o Grasp possui dois problemas relatados pelos pesquisadores. O primeiro é o custo computacional com a manutenção de toda a estrutura do grafo de implicações. Caso as estruturas não sejam construídas de forma eficiente, o tempo necessário para a atualização do grafo pode atrapalhar o desempenho do resolvedor.

O segundo problema se refere ao tamanho e ao número de cláusulas inseridas na fórmula a cada retrocesso. Sem um controle, elas podem crescer exponencialmente com relação ao número de variáveis da fórmula. Uma maneira de solucionar esse problema é ser mais seletivo com as cláusulas introduzidas na fórmula. Para isto, o Grasp define um valor arbitrário como parâmetro do resolvedor, cláusulas com número de literais menor que esse valor, são tratadas como descrito anteriormente. As cláusulas com número de literais superior ao valor definido só devem ser mantidas na fórmula enquanto forem cláusulas unitárias, e descartadas quando deixarem de ser. Apesar desta heurística de gerenciamento de cláusulas ser eficaz para o problema, muita informação relevante pode ser descartada com as cláusulas removidas. Desta forma alguns resolvedores posteriores ao Grasp apresentam novas formas de gerenciar as cláusulas aprendidas através de conflitos. O mais representativo desses resolvedores é o Chaff, apresentado a seguir.

3.4 Melhorias introduzidas pelo Algoritmo Chaff

Baseado no algoritmo Grasp, com a mesma estrutura herdada do DPLL e utilizando técnicas semelhantes de aprendizado sobre conflitos, o resolvedor Chaff[8] introduziu modificações capazes de melhorar o desempenho dos resolvedores da época.

As principais modificações introduzidas são as novas heurísticas para a escolha da variável sobre a qual será executada a regra de ramificação, e um novo procedimento para a construção das cláusulas aprendidas. Políticas diferenciadas sobre a manutenção de

novas cláusulas na fórmula e sobre o reinício do procedimento de resolução também foram apresentadas, assim como otimizações no procedimento de localização e manipulação de cláusulas unitárias.

A cada nova decisão realizada pelo resolvedor, é necessário verificar se alguma cláusula unitária foi gerada e localizar seu literal ainda não valorado para expandir a implicação gerada. Como o número de decisões realizadas por um resolvedor é alto, esse processo de localização acaba tomando grande parte do tempo total de processamento. Visando reduzir isso, o Chaff utiliza um procedimento que visita apenas as cláusulas que se tornaram unitárias recentemente. Cada cláusula possui um contador, atualizado a cada decisão, que mostra o número de literais valorados como falsos na cláusula. Assim a cláusula é visitada apenas quando esse contador for igual ao número total de literais da cláusula menos um.

Abstraindo essa idéia, e se valendo do que já foi apresentado pelo algoritmo SATO, o resolvedor utiliza apontadores para monitorar dois literais quaisquer da cláusula que ainda não tenham sido decididos. Enquanto esses literais monitorados não tiverem um valor-verdade atribuído, a cláusula não será unitária e o algoritmo não poderá realizar nenhuma ação com ela no momento. Se qualquer literal for decidido como verdadeiro a cláusula foi resolvida e o monitoramento perde o sentido. Caso um desses literais seja valorado negativamente ele deixará de ser monitorado e um outro literal não valorado deverá ser escolhido para o seu lugar. Quando não existir outro literal não valorado, essa será uma cláusula unitária e o literal que deve ser implicado está sendo monitorado pelo outro apontador.

Entre os principais resolvedores, várias heurísticas são utilizadas para a escolha de qual proposição deve ser valorada pela regra de ramificação. A mais simples dessas estratégias é aquela que escolhe de forma aleatória uma variável do conjunto de todas as que ainda não foram valoradas. Outra técnica utilizada escolhe a variável que maximiza alguma função com relação a valoração atual e o conjunto de cláusulas da fórmula. Ainda existe uma terceira estratégia utilizada que escolhe o literal que aparece com mais frequência no conjunto de cláusulas não resolvidas.

Apesar de cada uma ter seus prós e contras, não existe atualmente uma métrica eficaz para definir qual a melhor estratégia a ser utilizada. Desta maneira, os pesquisadores do Chaff apresentaram uma nova heurística, com bons resultados obtidos nos testes realizados[8]. Para cada literal, positivo ou negativo, um contador é atribuído. A cada nova cláusula inserida na fórmula, o algoritmo incrementa o contador de todos os literais presentes na nova cláusula. O literal com o maior valor neste contador deve ser o escolhido para a nova decisão. Em caso de empate a escolha é feita de forma aleatória. Periodicamente, todos os contadores são divididos por uma constante. Esta estratégia é direcionada para a resolução de problemas considerados difíceis, aqueles onde ocorre um grande número de conflitos, visto que os contadores apenas armazenarão informações referentes as variáveis envolvidas em conflitos.

Apesar de muito parecida com a estratégia do algoritmo GRASP, de análise do grafo de implicação para o aprendizado de novas cláusulas, o algoritmo Chaff apresenta uma heurística diferenciada, denominada Primeiro UIP. Esta mesma estratégia já foi adotada por versões mais recentes do Grasp. UIP é a sigla em inglês para Ponto de Implicação Único.

Um vértice é dito dominante sobre um outro vértice qualquer do grafo de implicação quando todos os caminhos, a partir da primeira decisão do mesmo nível do vértice dominante, precisam passar por ele para alcançar o vértice dominado. Um UIP é um vértice que, dentro do seu nível de decisão, domina o vértice que descreve o conflito encontrado. Enumerando os UIPs a partir do vértice de conflito, o primeiro UIP é aquele que está mais próximo do conflito.

No grafo de implicação da figura 3.4, as implicações das decisões das váriaveis A e B acabaram gerando um conflito sobre a váriavel E. O nível de decisão do conflito é igual a 2 e a primeira decisão desse nível é aquela sobre a váriavel B. Partindo do nodo que representa a decisão de B, todos os caminhos que levam até o nodo que representao conflito, K, passam pelos nodos C e D. Dessa maneira, C e D são UIPs, e D é dito primeiro UIP por estar mais perto do conflito.

A estratégia utilizada no Chaff difere daquela apresentada no Grasp no momento em

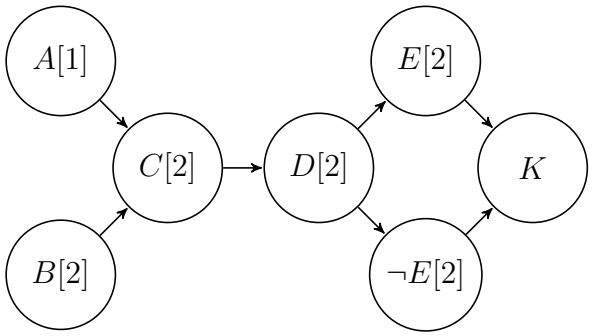

Figura 3.5: Grafo de Implicação com Conflito e Níveis de decisão

que é montado o conjunto de decisões antecessoras ao conflito. Ao invés de substituírmos cada vértice pelos seus antecessores até que sobrem apenas vértices que representem decisões não provenientes de uma implicação, o primeiro UIP de cada nível de decisão, é mantido no conjunto. Nenhum vértice antecessor de um UIP deverá ser adicionado. Desse ponto em diante os procedimentos para a geração e utilização da cláusula aprendida são semelhantes. Outras heurísticas baseadas em UIP são possíveis. Algumas definem que o segundo, e não o primeiro UIP devem ser mantidos no conjunto de vértices responsáveis por um conflito. Outras heurísticas defendem que o último UIP deve ser armazenado. Porém, em geral, a técnica do primeiro UIP é a mais difundida.

Outra modificação introduzida diz respeito à manutenção de cláusulas aprendidas na fórmula. Ao invés de mantê-las enquanto forem cláusulas unitárias, como faz o Grasp, o Chaff atribui a cada cláusula um mesmo valor arbitrário. A cláusula só será removida quando o número de literais não valorados atingir esse valor. Dessa maneira o algoritmo visa manter dentro do conjunto total de cláusulas aquelas que foram aprendidas e que estão, dependendo do valor arbitrário adotado, próximas de se tornarem cláusulas unitárias ou vazias.

A última diferença introduzida pelo Chaff é a possibilidade de reiniciar o processo de busca, apagando todas as decisões tomadas mas mantendo as cláusulas aprendidas. Essa técnica, aliada a uma escolha aleatória sobre qual literal valorar inicialmente, dá ao Chaff algumas vantagens encontradas nos algoritmos baseados em busca local, como o GSAT.

3.5 O Algoritmo GSAT: Satisfatibilidade utilizando Busca Local

Através da apresentação, em 1992, do resolvedor conhecido como GSAT, por Bart Selman, Hector Levesque e David Mitchell[11], uma nova linha de desenvolvimento foi adicionada às pesquisas relacionadas à satisfatibilidade: os resolvedores baseados em busca local. Estes resolvedores se caracterizam por, a partir de uma valoração inicial completa, realizar alterações sobre os valores decididos para cada variável de forma que a próxima valoração testada se mantenha próxima à atual.

Comparados aos resultados obtidos com o algoritmo DPLL, especialmente quando executado sobre fórmulas geradas aleatoriamente, estes resolvedores apresentam resultados competitivos[11], apesar de não serem algoritmos completos. Mesmo que a fórmula seja satisfatível este procedimento pode ser incapaz de encontrar uma valoração que torne a mesma verdadeira.

O resolvedor GSAT inicia com uma interpretação arbitrário da fórmula. Caso esta valoração não seja um modelo, alguma decisão precisa ser revista e alguma variável deverá ter seu valor-verdade alterado. Para realizar a escolha de qual variável deve ser alterada, o algoritmo atribui a cada uma um peso. Esse peso corresponde ao número de cláusulas que estão atualmente valoradas como falsas e que, a partir da troca do valor-verdade da variável, passarão a ser consideradas verdadeiras. A variável escolhida será aquela com maior peso. Em caso de empate entre as variáveis, a escolha pode ser realizada ao acaso. Ou seja, a função objetivo desse algoritmo de busca local pode ser considerada aquela que minimiza o número de cláusulas falsas de uma fórmula lógica proposicional. Este procedimento de troca se repete até que a valoração testada torne a fórmula verdadeira.

Sendo um algoritmo baseado em busca local, pois cada valoração testada dentro do espaço de busca difere da anterior em apenas uma variável, a execução do procedimento descrito pode levar a um problema conhecido como mínimo local. Guiando suas alterações pela busca do menor número de cláusulas falsas e partindo de uma valoração inicial aleatória, o resolvedor pode chegar a uma valoração onde todas as adjacentes possuem o número de cláusulas falsas maior que o atual, sendo impossível para o algoritmo decidir qual alteração realizar.

É possível também que o algoritmo acabe realizando trocas cíclicas em um conjunto restrito de variáveis. Para solucionar esse problema um número máximo de trocas deve ser definido como parâmetro do algoritmo. Quando o número de trocas realizadas atinge o máximo definido, o procedimento reinicia a busca a partir de uma nova valoração aleatória. Usualmente, esse número máximo deve ser algumas poucas vezes maior que o número de variáveis da fórmula. Apesar da eficiênica do reinício para solução do problema de mínimo local, testes demonstram que dificilmente um modelo será encontrado com a adoção desta técnica.

A solução adotada pelo GSAT para escolher sobre qual variável realizar a troca quando nenhuma delas reduz o número de cláusulas atualmente falsas, é escolher ao acaso uma, dentre aquelas que não incrementam o número de cláusulas falsas. Esta heurística, apesar de ser contrária a idéia da busca local, se mostra eficiente para a resolução do problema.

Para fórmulas não satisfatíveis, um número máximo de reinícios precisa ser definido no conjunto de parâmetros do algoritmo, caso contrário, o algoritmo continuará a sua execução indefinidamente. Porém, para uma fórmula lógica proposicional satisfatível, se o número máximo de reinícios for alcançado, e nenhum modelo for obtido, o resolvedor irá considerar erroneamente a fórmula como não satisfátivel. Esse número máximo pode ser tão grande quanto o tempo de execução disponível, porém, pela aleatoriedade dos reinícios não é possível garantir que todo o espaço de busca será pesquisado, confirmando a não completude do procedimento.

O procedimento pode ser representado pelo Algoritmo GSAT apresentado no algoritmo 3.2, sendo F uma fórmula na FNC, MR o número máximo de reinícios e MT o número máximo de trocas.

Algoritmo 3.2: Algoritmo GSAT

```
1 GSAT(F, MR, MT)
2     para I := 1 até MR faça
3         INTERPRETACAO:= ATRIBUIÇÃO_ALEATÓRIA(F);
4
5         para J := 1 até MT faça
6             se INTERPRETACAO satisfaz F
```

```
7                então retorne INTERPRETACAO;
8
9            INTERPRETACAO := TROCA_VALORAÇÂO(INTERPRETACAO);
```

3.6 Melhorias introduzidas pelo Algoritmo WalkSAT

Durante uma execução normal do algoritmo GSAT, o número de cláusulas valoradas como falsas na fórmula tende a decair. Porém, entre essas reduções, este número pode se manter constante durante um período da execução. A capacidade de sair destas planícies do espaço de busca, é o principal fator para determinar a eficiência de um resolvedor baseado em busca local. Porém, a técnica de escolher variáveis que não aumentem o conjunto de cláusulas não satisfeitas, usado pelo GSAT, não garante que o resolvedor será capaz de sair de uma dessas situações, contornando desta forma o problema de mínimo local. O resolvedor pode cair em um conjunto de valorações próximas a atual, onde o número de cláusulas não satisfeitas é constante, e nenhuma é um modelo da fórmula. Uma solução para contornar este problema é permitir a realização de trocas sobre variáveis que incrementem o número de cláusulas não resolvidas. Para a escolha dessa variável duas heurísticas são apresentadas a seguir.

A primeira é formulada utilizando um modelo de ruído baseado em mecanismos estatísticos. Após calculado o peso de cada uma das variáveis da fórmula, será escolhida aquela que diminui o número de cláusulas falsas, assim como no procedimento GSAT. Caso não exista uma variável capaz de reduzir, com a alteração do seu valor-verdade, o número de cláusulas falsas da fórmula, será escolhida uma aleatoriamente, com probabilidade igual a $e^{\frac{-p}{T}}$. Onde p é o peso da variável e T é um parâmetro do resolvedor, denominado temperatura. Esta temperatura pode ser constante ou ser decrescente durante o processo. Usualmente uma redução geométrica é realizada sobre esse parâmetro, repetidamente reduzindo-o com a multiplicação de uma constante menor que um. Quanto menor a temperatura, mais chances que uma variável com peso elevado, possa ser escolhida.

A segunda heurística propõe, a partir de uma valoração, também, gerada aleatoriamente, que seja escolhida uma cláusula da fórmula. Esta cláusula deve ser escolhida dentro do conjunto de todas aquelas que não são satisfatíveis com a valoração atual. Entre os literais presentes na mesma, algum deve ser escolhido ao acaso e seu valor-verdade alterado, satisfazendo a cláusula. Com esta cláusula satisfeita o procedimento se repete para as demais ainda não satisfeitas. Essa simples heurística, apesar de efetiva para alguns grupos de fórmulas, se mostra ineficiente para a resolução do problema de satisfatibilidade, de forma geral. Desta forma Bart Selman, Henry Kautz e Bram Cohen apresentaram, em 1995, uma terceira heurística, introduzida no algoritmo WalkSAT[10].

Fortemente atrelado à estrutura do GSAT, o WalkSAT difere apenas no processo de escolha da váriavel que deverá ter seu valor trocado. Com uma probabilidade p, a variável será escolhida dentre aquelas presentes em uma das cláusulas não satisfeitas da fórmula, também escolhida ao acaso. Esse procedimento é o mesmo da segunda heurística descrita anteriormente. Nos passos seguintes, que ocorrerão com probabilidade igual a $(1 - p)$, a variável será escolhida aplicando-se as mesmas heurísticas utilizadas pelo algoritmo GSAT. Apesar de simples, essa mudança aumenta consideravelmente a taxa de acerto sobre a satisfatibilidade de uma fórmula do resolvedor WalkSAT, com relação ao GSAT.

CAPÍTULO 4

SATISFATIBILIDADE EM FNN

Como descrito anteriormente, os esforços empregados na pesquisa de procedimentos capazes de solucionar a questão da satisfatibilidade de uma formula lógica tem se concentrado principalmente sobre fórmulas que se encontram na FNC. Isso se explica, em grande parte, pelas facilidades que esse formato apresenta para muitas das técnicas estudadas, como a Propagação de Restrições Unitárias. Porém, modelar problemas reais, como a verificação de circuitos lógicos, utilizando fórmulas lógicas em FNC se mostra uma tarefa complexa. Em geral esses problemas são mais facilmente representados em fórmulas não clausais que, quando restritas aos mesmos operadores lógicos utilizados na FNC, se encontram na Forma Normal Negada, ou FNN. Uma fórmula está na FNN quando é composta, exclusivamente, por conjunções e disjunções, sem uma hierarquia definida entre os operadores, e as negações então presentes apenas nos literais. Fórmulas na FNN são, geralmente, mais sucintas que fórmulas equivalentes na FNC.

Converter uma fórmula na FNN para a FNC pode aumentar sua complexidade, seja adicionando novas variáveis ou com a perda de informações referentes a estrutura inicial da fórmula. O aumento dessa complexidade pode ser crucial para a eficiência dos procedimentos que visam definir a satisfatibilidade da fórmula.

O método mais prático de conversão é o que introduz novas proposições na fórmula. Esse procedimento resulta em uma fórmula equivalente e com aumento linear de tamanho[9]. Uma outra maneira, é a aplicação das propriedades distributivas dos conectivos de conjunção e disjunção, como já foi anteriormente demonstrado. Apesar da fórmula resultante possuir o mesmo conjunto de variáveis, o seu tamanho pode ser exponencialmente maior em relação ao problema inicial.

Levando em conta as dificuldades de ambos os métodos, algumas pesquisas buscam novos métodos, ou a adaptação de métodos já existentes utilizados em fórmulas na FNC,

para a solução de problemas representados na FNN, retirando assim a necessidade de transformação da fórmula. Alguns dos procedimentos mais conhecidos serão detalhados a seguir.

4.1 Algoritmo baseado em DPLL para FNN utilizando Grafos

Como apresentado nos capítulos anteriores, os principais processos para a resolução do problema de satisfatibilidade em fórmulas lógicas são baseados nos conceitos apresentados pelo algoritmo DPLL. Os três principais passos do DPLL, remoção de literais puros, propagação unitária e aplicação da regra de ramificação, são as atividades básicas dos algoritmos que derivam desta técnica. Entre os três passos, o único cuja aplicação é dependente do formato em que a fórmula está representada é também o que demanda o maior tempo computacional durante a resolução do problema, aquele responsável pela propagação unitária. O próprio conceito da atividade, assim como as diversas melhorias técnicas introduzidas nas várias pesquisas realizadas nos últimos anos, estão fortemente atreladas à Forma Normal Conjuntiva. Sendo assim, a criação de um procedimento para a resolução do problema de satisfatibilidade para formas não clausais, baseado em DPLL, depende do desenvolvimento de uma técnica eficiente para a aplicação da propagação unitária, visto que essa etapa do processo se repete várias vezes e onde as técnicas conhecidas são mais dependentes do formato de representação utilizado para a fórmula lógica. Essa é a proposta do método apresentado na pesquisa de Himanshu Jain e Edmund M. Clarke[7], um resolvedor SAT eficiente para fórmulas na FNN. Esse procedimento utiliza uma representação da fórmula com grafos, visando facilitar a propagação unitária durante o processo.

O algoritmo apresentado opera sobre problemas descritos em fórmulas lógicas na FNN, ao contrário de outros métodos não clausais que utilizam a representação em circuitos lógicos. Em uma representação baseada em circuitos, trechos da fórmula podem ser compartilhados, ou seja, uma sub fórmula pode ser utilizada como operando de dois ou mais conectivos lógicos. Para remover esse compartilhamento, o algoritmo utiliza uma técnica de inserção de variáveis. Cada conectivo raiz de uma sub fórmula, utilizada por

Figura 4.1: Grafo hpgraph para Literais, Conjunções e Disjunções

mais de um conectivo, é substituído por uma nova variável lógica ainda não presente na fórmula. Assim que todos os trechos compartilhados forem removidos da fórmula, as possíveis negações são propagadas à sub fórmula com a aplicação das equivalências lógicas apresentadas no capítulo 2. Apesar desse método aumentar o número de variáveis lógicas a serem valoradas no problema inicial, estudos mostram que esse número ainda é menor do que aquele presente na fórmula equivalente na FNC[7].

O grafo denominado hpgraph, utilizado para representar a fórmula, pode ser descrito por quatro conjuntos: o primeiro é o conjunto de todos os vértices, seguido pelo conjunto dos vértices raiz, o dos vértices que são folha e o das arestas. Cada literal presente na fórmula será representado por um único vértice no grafo final. Se um vértice não possui nenhuma aresta incidindo sobre ele, será denominado como um vértice raiz. No caso de não possuir nenhuma aresta com origem nele, será denominado folha.

A criação do grafo hpgraph, a partir de uma fórmula lógica, segue um conjunto definido de regras. O grafo, para um literal qualquer, será um novo vértice, que representará aquele literal. Já o grafo para uma conjunção será a união dos grafos de todos os operandos da sub fórmula. Por fim, o grafo representante de uma disjunção é obtido ligando, por arestas, cada vértice folha do grafo do primeiro operando com cada vértice raiz do grafo do segundo.

Na figura 4.1 é apresentado, respectivamente, os grafos hpgraph para as fórmulas (A), $(A \wedge B)$ e $(C \vee D)$.

Com base no grafo hpgraph resultante, é possível obter uma fórmula equivalente à original, porém, na FNC. Partindo de cada vértice raiz do grafo, cada caminho completo que termine em um vértice folha, representa uma cláusula na FNC. Essa cláusula é formada por todos os literais representados por vértices presentes nesse caminho. Com o conjunto de todos os caminhos do grafo, todas as cláusulas da fórmula lógica proposicional na FNC

podem ser obtidas.

A partir de uma valoração, parcial ou completa, das variáveis da fórmula, se todos os vértices que formam um caminho forem literais valorados como falso, a cláusula representada pelo caminho será falsa. Isto é suficiente para definir que a valoração não representa um modelo da fórmula. Essa característica do grafo hpgraph é a base das técnicas do algoritmo DPLL para formas não clausais.

Uma das principais técnicas para a propagação unitária em fórmulas na FNC é a utilização de variáveis monitoradas, como descritos nos capítulos anteriores nas seções que tratam dos algoritmos SATO e Chaff. A maneira apresentada de utilizar o conceito de literais monitorados em uma fórmula representada por um grafo hpgraph é a partir de cortes no próprio grafo, ou um sub conjunto dos vértices do grafo. Mantendo dois cortes, onde cada um contenha ao menos um vértice pertencente a cada caminho presente no grafo, ou a cada cláusula da fórmula na FNC, é possível obter o equivalente aos dois literais monitorados em uma cláusula na FNC. É importante que os cortes não compartilhem vértices entre si. O procedimento segue a mesma estrutura daquele realizado nos algoritmos para FNC. Cada vez que um vértice de um corte for valorado como falso, devemos substituí-lo. Caso a substituição não seja possível, obtemos ou um conflito ou uma implicação a ser utilizada com a propagação unitária.

A principal contribuição da pesquisa realizada é a utilização de um grafo vpgraph para a geração dos cortes no grafo hpgraph. Esse grafo vpgraph é gerado a partir da fórmula do problema inicial com regras similares a do hpgraph. A regra para a geração do grafo a partir de um literal é a mesma, porém, inversamente ao que ocorre com o hpgraph, no caso da disjunção o grafo resultante será representado pela a união dos grafos de todos os operandos. Já a conjunção será a ligação, por arestas, dos vértices folha do primeiro operando com os vértices raiz do segundo.

Na figura 4.1, os grafos vpgraph para as fórmulas (A), $(A \wedge B)$ e $(C \vee D)$ são apresentados.

No grafo vpgraph, o caminho entre um vértice raiz e um vértice folha representa uma conjunção de uma fórmula na FND, Forma Normal Disjuntiva, representada pela

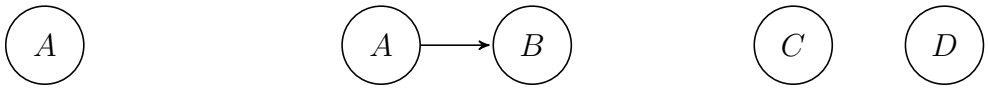

Figura 4.2: Grafo vpgraph para Literais, Conjunções e Disjunções

disjunção de conjunções. Cada uma dessas conjunções representa um corte mínimo necessário para o monitoramento descrito anteriormente. Dessa forma, o problema de encontrar novos cortes para substituir os anteriores durante o processo de busca do DPLL, pode ser resolvido com uma busca em profundidade nos caminhos do grafo vpgraph gerado. Implementando dessa forma a propagação unitária sobre uma fórmula não clausal, todos os outros passos do algoritmo DPLL podem ser utilizados em uma fórmula que se encontra na FNN.

4.2 O Algoritmo NoClause

Apesar da simplicidade e da eficiência do procedimento DPLL para uma fórmula não clausal, com a utilização de grafos, ele apenas reduz os principais problemas dos resolvedores que transformam as fórmulas iniciais para a FNC. A informação estrutural do problema original, perdida durante esta transformação, pode servir como fonte de informação para as heurísticas utilizadas pelo resolvedor. Mas a principal restrição fica por conta da impossibilidade do resolvedor de tratar fórmulas apresentadas como circuitos, com a reutilização de subfórmulas, principal formato utilizado pelos problemas da área de testes de hardware. Apesar desse fato, de forma isolada, não garantir um aumento no tempo de resolução do problema, os recursos utilizados na conversão poderiam ser aplicados em outros pontos do resolvedor. Visando mostrar que a conversão de uma fórmula de um formato não clausal para a FNC é, ao mesmo tempo, desnecessária e indesejada, Christian Thiffault, Fahiem Bacchus e Toby Walsh, apresentaram o NoClause[14], um algoritmo que aplica DPLL sobre uma fórmula na FNN e visa tirar vantagem de suas informações estruturais.

Ao contrário do formato tradicional, utilizado para a representação computacional de uma fórmula lógica, o NoClause utiliza um grafo direcionado acíclico, em oposição a

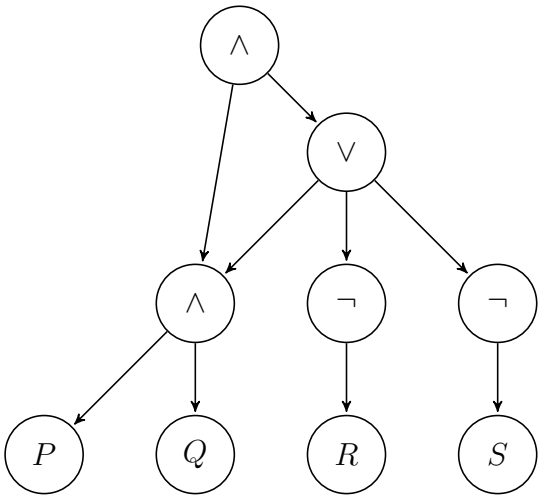

Figura 4.3: Grafo de Representação Utilizado pelo Algoritmo NoClause

estrutura de árvore, para guardar as informações do problema. Cada vértice do grafo representa ou um literal ou um operador lógico da fórmula. As relações entre esses elementos são representadas na forma de arestas direcionadas, partindo dos operadores e incidindo sobre os operandos. Para endereçar de maneira fácil essa questão, cada vértice guarda duas listas, uma com todos os vértices pais, aqueles que estão acima do mesmo na hierarquia da fórmula, e outra com os vértices filhos, aqueles sobre os quais ele opera. Cada literal da fórmula aparece uma única vez no grafo descrito, ainda que seja utilizado múltiplas vezes na fórmula. Dessa maneira a representação de um problema na forma de um circuito se torna mais fácil. O vértice também deve guardar informações sobre o valor-verdade atribuído a ele e qual a causa de tal atribuição, se por inferência ou decisão do resolvedor, visando assim facilitar possíveis retrocessos.

A figura 4.2 é apresenta o grafo utilizado pelo algoritmo NoClause para a representação interna da fórmula lógica $((P \wedge Q) \vee \neg R \vee \neg S) \wedge (P \wedge Q)$.

Com esse grafo gerado, o objetivo do algoritmo é atribuir uma valoração consistente para o conjunto de vértices, visando encontrar um modelo para a fórmula, de forma que o vértice raiz, aquele sobre o qual nenhum outro vértice opera, seja definido como verdadeiro. Uma valoração consistente é aquela onde os valores atribuídos aos vértices respeitam a semântica dos operadores da fórmula.

O algoritmo inicia o procedimento atribuindo a algum vértice não valorado, um valor-

verdade. A seguir, esse valor é propagado para seus pais e filhos, conforme a necessidade. Por exemplo, se o vértice for valorado como verdadeiro, cada um de seus vértices pais que for uma disjunção poderá ter seu valor implicado como verdadeiro. O processo ocorre de forma análoga com os pais formados por conjunções, quando o vértice é valorado como falso.

Para mantermos uma ordem cronológica sobre as decisões tomadas pelo resolvedor, cada vértice deve armazenar o nível de decisão em que lhe foi atribuído o valor-verdade. Esse nível de decisão é incrementado a cada novo ciclo, tal qual o nível de decisão presente no grafo de implicações do algoritmo GRASP. Um conflito ocorre quando, ao mesmo tempo, um vértice recebe valores complementares. Nesse caso é necessário retroceder alguma decisão anterior e continuar o processo.

Baseado na estrutura adotada no grafo, o algoritmo pode utilizar técnicas como a do primeiro UIP, apresentada no algoritmo Chaff. No caso de um conflito, é possível retroceder entre os vértices responsáveis pela inferência da decisão equivocada até que seja encontrada a decisão que deverá ser revista. Durante esse processo, cláusulas, que descrevem o conjunto de decisões capazes de gerar um conflito, são aprendidas e armazenadas em uma fórmula FNC. Monitorando essa fórmula o algoritmo evita a repetição de erros. Essa fórmula na FNC, criada para armazenar as cláusulas aprendidas através de conflitos, sofre dos mesmos problemas de gerenciamento, no que se refere ao tamanho das cláusulas aprendidas, que já foi discutido nos algoritmos GRASP e Chaff.

Para reduzir a necessidade de verificar uma grande quantidade de vértices a cada decisão tomada, na busca por novas inferências, o NoClause utiliza a técnica de monitorar alguns vértices. Um vértice que represente um conectivo de conjunção só pode ser inferido como verdadeiro quando todos os seus vértices filhos forem decididos como verdadeiros. Se, a todo momento, forem monitorados dois vértices filhos dessa conjunção, que não tenham sido valorados, não se torna necessário visitar os demais filhos. No instante que algum dos vértices monitorados for valorado como verdadeiro, é preciso substituí-lo por um outro filho ainda não decidido. Caso a substituição não seja possível o vértice pai deve ser inferido como verdadeiro. Caso ele ainda não tenha um valor definido, e o vértice

pai já tenha sido decidido como verdadeiro, o vértice monitorado e não valorado pode ser inferido como verdadeiro também. O mesmo princípio se aplica a vértices que representem disjunções, onde o algoritmo irá buscar filhos não valorados ou decididos como falso.

Com essa implementação, o NoClause mostra que é possível resolver problemas de satisfatibilidade utilizando técnicas bem conhecidas e difundidas, sem se prender a fórmulas em um formato restritivo, como a FNC.

4.3 O Algoritmo PolSAT

Os algoritmos para satisfatibilidade baseados em busca local são, possivelmente, os mais simples de serem adaptados à representações não clausais. Em geral, esses algoritmos apresentam duas funções básicas. A primeira define, dentro do conjunto total de variáveis da fórmula, quais serão aquelas que podem ter o seu valor-verdade alterado. A outra função escolhe, neste conjunto, qual variável deve ter o seu valor alterado. Essa alteração modifica a valoração atual de tal forma que a nova valoração esteja mais próxima de um modelo da fórmula. Em outras palavras, escolhe-se a alteração que obtenha um melhor resultado junto a função objetivo da busca local[9].

Partindo de uma valoração inicial, normalmente arbitrária, esse ciclo de escolha de variáveis proposicionais e troca de seus valores-verdade se repete um número finito de vezes, ou até que uma interpretação que satisfaça a fórmula seja encontrada. Em momentos predeterminados, o algoritmo pode abandonar a valoração atual e recomeçar o processo com uma nova valoração aleatória, mantendo assim, os conceitos de parâmetros que definam números máximos de trocas e de reinicios, como apresentado no algoritmo GRASP.

Todos esses passos independem do formato adotado para a representação da fórmula, com exceção ao método capaz de avaliar quão distante de uma solução uma valoração está, a função objetivo. Definir procedimentos capazes de realizar, de forma eficiente, o processo de escolha da variável a ser alterada é o principal desafio de um algoritmo baseado em busca local, para o problema de satisfatibilidade em fórmulas não clausais. Sobre essa questão, os desenvolvedores do algoritmo PolSat[13] apresentam algumas soluções.

O PolSat utiliza a mesma estrutura de passos descrita no algoritmo WalkSAT. Para gerar a lista de possíveis variáveis candidatas a terem o seu valor-verdade alterado, o algoritmo utiliza um peso atribuído a cada elemento da fórmula. Esse peso é calculado de forma recursiva, dos literais da fórmula para os conectivos que operam sobre eles. O peso atribuído a um literal será igual a um, se na valoração atual ele for falso, caso contrário, seu peso será zero. Para uma disjunção, o seu peso será igual ao menor peso entre todos os seus operandos. No caso da conjunção, a soma do peso de todos os operandos será o seu peso final. Basicamente, esse peso tenta descrever quantos elementos valorados como falso se encontram abaixo daquele conectivo, ou quão falso ele é na valoração atual.

O procedimento de cálculo da lista de variáveis candidatas inicia inserindo em uma lista o conectivo raiz da fórmula. Cada conectivo dessa lista é substituído pelo seu operando que possuir o menor peso. Em caso de empate ambos são inseridos. O processo se repete até que sobrem apenas literais. A lista final será formada por todas as variáveis representadas pelos literais encontrados. Com uma probabilidade p a variável alterada será escolhida a partir da lista de candidatas de forma aleatória. Com uma probabilidade $1-p$ será escolhida aquela cuja alteração reduz mais significativamente o peso acumulado da fórmula, representado pelo peso do conectivo raiz.

Um dos principais problemas enfrentados por algoritmos baseados em busca local, além do fato de sua não completude, é começar a realizar trocas cíclicas entre as variáveis da fórmula, sendo incapaz de achar um modelo para a mesma. O aprendizado de cláusulas, utilizado por vários outros resolvedores, pode garantir que uma interpretação não seja testado repetidamente. Gerar essa cláusula sobre as váriaveis presentes na lista de candidatas a terem seu valor alterado foi o caminho adotado nas primeiras versões do PolSat. Para cada variável da lista, se for considerada verdadeira na valoração atual, a inserimos na cláusula aprendida como um literal negativo. Caso ela apareça falsa na valoração testada, será inserida na cláusula resultante como um literal positivo. Essa lista de cláusulas aprendidas pode ser inserida no problema original através de uma conjunção entre a fórmula inicial e as cláusulas.

Outro ponto onde este aprendizado pode auxiliar o desempenho do resolvedor é na

heurística de decisão sobre qual váriavel, da lista das candidatas, realizar a alteração. Como visto anteriormente, ou essa escolha será aleatória ou será por aquela que mais reduzir o peso acumulado. Em caso de duas varáveis empatarem na taxa de redução do peso acumulado, o algoritmo pode decidir por aquela que tornar falsas o menor número de cláusulas aprendidas.

CAPÍTULO 5

PLATAFORMA PARA COMPARAÇÃO DE HEURÍSTICAS PARA SAT NÃO CLAUSAL

Nos capítulos anteriores, foi analisada a evolução dos principais algoritmos que visam resolver o problema de decidir sobre a satisfatibilidade de uma fórmula lógica proposicional. A maioria dos resolvedores existentes está dividida entre duas grandes classes, busca com retrocesso e busca local, e são baseados em um resolvedor principal, DPLL[4] ou GSAT[11], respectivamente. Cada nova solução apresentada se destaca por alterações das técnicas utilizadas por esses algoritmos principais[15, 12, 8, 10]. Pequenas alterações nos procedimentos conhecidos geraram procedimentos com desempenho superior ao de seus antecessores. Exemplos dessas alterações podem ser demonstrados com o monitoramento de apenas dois literais de uma cláusula, apresentado pelo SATO[15], ou com a adição de uma escolha arbitrária para o problema de decidir qual proposição deve ter seu valor trocado, do WalkSAT[10].

Em um problema onde os passos de um algoritmo resolvedor são repetidos inúmeras vezes, decisões sobre qual a próxima proposição a ser decidida ou que tipo de estruturas de dados permitem um acesso mais rápido a um literal da fórmula, possuem um forte impacto sobre o desempenho do processo. Para os algoritmos que visam tratar o problema de satisfatibilidade, mesmo aqueles que utilizem como base um resolvedor já existente, várias escolhas são possíveis, seja essa decisão sobre as diferentes heurísticas aplicadas nas estapas do resolvedor ou sobre as estruturas de representação interna. Dependendo da classe de problemas tratados, fórmulas lógicas geradas de maneira aleatória ou fórmulas utilizadas na verificação de circuitos lógicos, por exemplo, mais de uma combinação eficiente destes elementos é possível.

Para os algoritmos que operam sobre fórmulas na FNC, a evolução dos diversos resolvedores apresenta uma vasta base de estudo sobre as principais vantagens das várias

escolhas para essa classe de problemas. Visando facilitar o desenvolvimento de resolvedores baseados em fórmulas na FNC, em 2003, o pesquisador Niklas Een introduziu o MiniSAT[6]. MiniSAT é um resolvedor minimalista, de código aberto, desenvolvido para auxiliar pesquisadores e desenvolvedores no ínicio dos trablahos nessa área. Devido ao grande número de resolvedores existentes e a grande diversidade de implementações, o processo de estudo e comparação dessas soluções se torna complicado. Tendo um resolvedor base, com foco na facilidade de alteração, como o MiniSAT, facilita esse processo. Uma base única retira do processo de avaliação de um conjunto de soluções outros fatores, como linguagens e estruturas de implementação, fatores que influenciam no desempenho de um sistema e não são sempre considerados.

Para os algoritmos baseados em fórmulas não clausais, muito esforço tem sido realizado na adaptação das técnicas utilizadas em resolvedores FNC para esse formato. Porém, com a possibilidade de representação de novas classes de problemas em um formato menos restritivo, possivelmente, novas combinações eficientes de heurísticas e estruturas podem apresentar resultados satisfatórios.

A seguir, são listadas algumas das decisões dos algoritmos baseados no resolvedor DPLL onde, quando operando sobre fórmulas não clausais, o mapeamento de técnicas difundidas entre os resolvedores baseados em fórmulas clausais não é trivial.

- Armazenamento e acesso a informações referentes a cada elemento da fórmula;
- Monitoramento de vértices em busca de novas implicações possíveis;
- Escolha de qual proposição da fórmula deve ser decidida durante a regra de ramificação;
- Forma de armazenamento das cláusulas aprendidas em cada conflito;
- Heurísticas adotadas para a geração de cláusulas aprendidas.

Decidir qual a combinação de técnicas que devem ser aplicadas para o problema tratado é uma tarefa complexa e uma série de testes são necessários para comprovar a eficácia das decisões tomadas. Além do fato de mais de uma solução satisfatória ser possível, os

mesmo problemas presentes nos algoritmos baseados em fórmulas clausais estão presentes nas fórmulas não clausais: a grande diversidade de linguagens e de estruturas de implementações.

Visando facilitar o processo de desenvolvimento e análise de novos algoritmos resolvedores, é proposta, a seguir, uma plataforma para resolvedores em fórmulas não clausais. Uma plataforma pode ser definida como uma abstração que une trechos de código comuns, que colaboram para prover funcionalidades genéricas. O principal objetivo é permitir, de forma simples e robusta, que as mais diversas técnicas possam ser implementadas e comparadas sobre conjuntos de fórmulas lógicas não clausais. Para tanto, estruturas de dados capazes de suportar as principais técnicas conhecidas são apresentadas. Não há a pretensão de apresentar estruturas otimizadas para técnicas específicas, mas sim de trabalhar com aquelas capazes de tratar de maneira satisfatória o maior número de problemas possível.

5.1 A Plataforma

Como mencionado anteriormente, uma plataforma que visa facilitar a implementação e os testes do maior número possível de técnicas que auxiliem na resolução do problema de satisfatibilidade para fórmulas na FNN, deve fornecer um conjunto básico de estruturas e funcionalidades. Essas estruturas e funcionalidades devem, idealmente, implementar um resolvedor básico. Dessa maneira, qualquer etapa do processo pode ser alterada sem que para isso seja necessária a implementação completa de um novo resolvedor. Como resolvedor básico, a plataforma apresentada utiliza uma implementação do algoritmo DPLL para fórmulas não clausais.

Antes de iniciar uma discussão individual sobre as várias etapas de um resolvedor baseado no algoritmo DPLL para fórmulas não clausais, é necessário uma visão geral do processo básico que deve ser realizado pela plataforma proposta.

Em seções anteriores, uma série de problemas que apresentam como sub-problema ou que podem ser reduzidos à questão da satisfatibilidade de uma fórmula lógica foram apresentados. Dada a grande variedade de problemas abordados, especialmente quando

se trata daqueles que recaem sobre fórmulas na FNN, um formato de representação único deve ser adotado. Por se tratar de uma forma de representação capaz de modelar um conjunto mais complexo de problemas, os formatos utilizados para fórmulas baseadas em cláusulas não são suficientes para uma representação satisfatória. Assim sendo, a plataforma deve ser capaz de ler e armazenar problemas que utilizem o formato adotado. Esse formato deve ser de fácil utilização, para que, ao mesmo tempo, a modelagem não se torne um impecilho e a leitura possa ser feita de forma rápida e inequívoca. Afim de não restringir a utilização de nenhuma técnica, o máximo de informação do problema original deve ser mantido na representação interna da plataforma.

Executado o passo inicial, a leitura da fórmula lógica proposicional, o resolvedor deve preencher suas estruturas internas com as informações obtidas. As estruturas utilizadas devem ser capazes de manter todos os dados do problema de forma consistente e possibilitar o acesso aos mesmos de maneira prática e rápida. Por se tratar de uma plataforma que visa a implementação e análise das mais diversas técnicas, se faz necessário que tal estrutura seja capaz de atender as várias necessidades de cada nova implementação, sendo transparente ao desenvolvedor quando atender as necessidades e permitindo sua substituição quando necessário. É possível que estruturas auxiliares sejam necessárias para armazenar informações adicionais da execução do processo. A plataforma proposta deve possibilitar a utilização de estruturas básicas comuns, como listas e filas, agilizando o desenvolvimento e permitindo análises comparativas mais consistentes.

Após o preenchimento de todas as estruturas internas, um resolvedor baseado no algortimo DPLL deve focar na execução dos três passos básicos que caracterizam essa solução: a aplicação da regra do literal puro, da regra da cláusula unitária e da regra da ramificação. Com exceção da última regra mencionada, que depende exclusivamente de uma estrutura interna que permita o acesso e a ramificação de cada elemento da fórmula, as demais regras são fortemente ligadas ao conceito de cláusula. Abstraindo os conceitos por trás da execução das duas primeiras regras, basicamente, suas funções são a simplificação e geração de novas implicações. As três funcionalidades básicas, ramificação, simplificação e geração de novas implicações devem estar presente na plataforma em implementações

básicas. Será sobre a execução dessas três funcionalidades que muitas técnicas serão implementadas, e para tal a plataforma deve permitir que alterações sejam realizadas de maneira simples nessas funções.

5.2 Formato de Representação das Fórmulas Não Clausais

A representação de fórmulas lógicas não clausais não é tão simples quanto aquelas adotadas pelas fórmulas na forma normal conjuntiva. O formato DIMACS CNF[1] se apresenta como o padrão adotado pelos resolvedores baseados em fórmulas clausais. Esse formato consiste de uma simples listagem de todas as cláusulas da fórmula, relacionando, para cada uma delas, todos os literais que a compõem. Essa informação não é suficiente para representar uma fórmula não clausal.

O formato ISCAS[2] apresentado em 1985 no International Symposium on Circuits And System, inicialmente para descrever uma série de problemas definidos para exercícios comparativos na área de verificação automática de circuitos, pode ser utilizado para descrever problemas de satisfatibilidade de fórmulas não clausais.

O formato descreve uma fórmula lógica proposicional, não clausal, listando todos os elementos que a compõem, suas proposições e operadores lógicos. Para cada um deles, é atribuído um identificador único. Para cada operador da fórmula, uma lista de todos os literais, ou outros operadores, sobre os quais ele opera é apresentada. O operador definido como raiz da fórmula é aquele sobre o qual nenhum outro opera é descrito de forma diferenciada. Por ser um formato definido para a descrição de circuitos lógicos, algumas mudanças na nomenclatura utilizada são necessárias. As váriaveis proposicionais da fórmula são definidas como entradas do circuito descrito. Já o operador raiz é definido como saída do mesmo circuito. Na figura 5.2 apresentamos uma fórmula lógica na FNN representada no formato ISCAS.

Por se tratar de um formato já bem difundido, algumas técnicas para a leitura dos circuitos representados, são utilizadas em outras áreas de aplicação. Porém, a obtenção da fórmula lógica, no contexto do problema de satisfatibilidade, pode ser feita de forma simples a partir do operador raiz da fórmula, realizando uma busca pelos operadores

$$((P \wedge Q) \vee \neg R \vee \neg S) \wedge (P \wedge Q)$$

```
1   INPUT(P)
2   INPUT(Q)
3   INPUT(R)
4   INPUT(S)
5   OUTPUT(RAIZ)
6
7   NOT_R = NOT(R)
8   NOT_S = NOT(S)
9   AND_1 = AND(P, Q)
10  OR_1  = OR(AND_1, NOT_R, NOT_S)
11  RAIZ  = AND(OR_1, AND_1)
```

Figura 5.1: Fórmula Lógica na FNN no formato ISCAS

listados. Com base no operador raiz, sempre que um novo operador, ou seja, aquele que ainda não esteve presente anteriormente na fórmula, for encontrado na lista dos sub elementos do operador atual, deve-se percorrer a lista de elementos da fórmula para descobrir sobre quais elementos ele opera. Esse processo é repetido até que todos os elementos encontrados a partir do operador raiz, sejam definidos. Todos os elementos da fórmula que podem ser encontrados apartir de uma busca iniciada no operador raiz são definidos como alcançáveis.

Dependendo de como a lista de operadores a serem definidos é tratada, a varredura da fórmula pode ser feita em largura ou em profundidade. Se cada novo operador encontrado é imediatamente tratado, o processo é definido como uma varredura em profundidade. Porém, se cada novo operador é enfileirado, e o primeiro da fila, mais antigo a ser encontrado, tem a preferência no processo de busca, o processo é definido como uma varredura em largura. Dependendo da fórmula representada e do conjunto de informações armazenadas no processo de leitura da fórmula, esses métodos podem apresentar desempenho diferenciado. Porém, ambos sempre devem gerar a mesma fórmula.

Além de permitir que muitas informações sobre a fórmula sejam obtidas já nessa primeira etapa do processo, como definir onde e com qual frequência cada elemento aparece na fórmula, esse formato de representação apresenta outras vantagens. Todos os elementos, váriaveis proposicionais e operadores, que são descritos na fórmula mas que não são alcançáveis a partir do operador raiz já são automaticamente descartados. Isso evita que,

em etapas futuras, esses elementos sejam, desnecessariamente, considerados durante o processo.

Outra vantagem, herdada da representação da fórmula como um circuito lógico, é a possibilidade de que trechos que aparecem repetidamente na fórmula inicial, possam ser representados uma única vez na estrutura final. Grandes circuitos lógicos, normalmente, podem ser compostos de pequenos conjuntos de portas lógicas, reponsáveis por alguma tarefa específica, que são utilizados repetidas vezes. Caso estruturas semelhantes já sejam previamente conhecidas em uma fórmula não clausal, a utilização de identificadores únicos para cada operador da fórmula, permite que elas sejam facilmente encontradas e que não exista duplicação de elementos na representação final. A não duplicação desses elementos depende, também, da estrutura interna de armazenamento da fórmula.

No exemplo apresentado anteriormente, a conjunção identificada como "AND_1" aparece mais de uma vez no decorrer da fórmula. Caso essa fórmula fosse convertida para um formato clausal, essa estrutura seria convertida e repetida mais de uma vez, consumindo espaço de armazenamento interno. Utilizando o processo de leitura descrito anteriormente, na segunda vez que uma referência a "AND_1" fosse encontrada, ela não seria considerada como um novo elemento e, consequentemente, não seria duplicada.

5.3 Estrutura Interna de Armazenamento

A escolha mais natural para a estrutura de representação de uma fórmula não clausal é a estrutura de árvore. Essa estrutura é caracterizada por dois tipos de elementos: os nós e as arestas. Cada aresta representa a ligação entre dois nós da árvore. As arestas podem ser direcionadas, ou seja, partindo de um nó, aqui denominado pai, e incidindo sobre um segundo nó, denominado filho. Essa estutura de parentesco entre os nós de uma árvore define semanticamente a relação dos elementos representados por ela. Um nó é denominado nó folha quando ele não é considerado pai de nenhum outro nó da árvore, ou seja, nenhuma aresta parte dele e incide sobre qualquer outro nó da fórmula. Já um nó que não possui nenhuma aresta partindo de qualquer nó da árvore e incidindo sobre ele, é denominado um nó raiz. Um nó raiz é aquele que não é filho de nenhum outro nó da

$$((P \wedge Q) \vee \neg R \vee \neg S) \wedge (P \wedge Q)$$

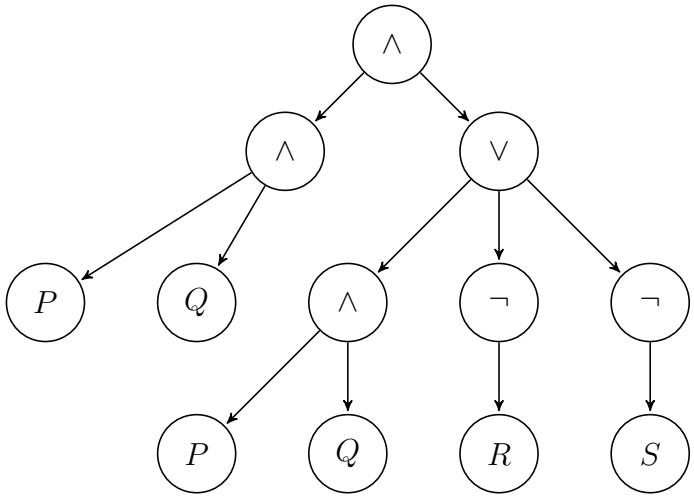

Figura 5.2: Fórmula Lógica Não-Clausal representada em uma estrutura de Árvore

árvore.

No contexto das fórmulas não clausais, cada elemento da fórmula, operador lógico ou váriavel proposicional, é representado por um nó da árvore. Cada nó que representa um operador está ligado através de arestas que partem dele e incidem sobre os nós que representam seus operandos. Dessa maneira, cada proposição da fórmula é representada por um nó folha da árvore. O nó raiz da árvore representa o operador raiz da fórmula. Dessa forma, partindo do nó raiz e percorrendo todas as arestas da árvore até os seus nós folhas, é possível percorrer a fórmula lógica em sua totalidade.

Na figura 5.3 demonstraremos a representação da fórmula lógica não clausal, apresentada no exemplo anterior, em uma estrutura de árvore.

Como descrito na seção anterior, a ocorrência de trechos repetidos na fórmula lógica é permitida e até mesmo provável quando o problema estudado for um dos obtidos pela modelagem de um circuito. No exemplo anterior, a estrutura que representa a sub-fórmula $P \wedge Q$ teve que ser duplicada. Apesar de ser representada no formato ISCAS como um único operador, identificado como "AND_1". Essa repetição não representa um consumo desnecessário de memória, quando ocorre sobre trechos pequenos, como no exemplo anterior. Porém, quando o algoritmo precisa resolver fórmulas com milhares de proposições e milhares de operadores lógicos, trechos repetidos podem tornar o sistema incapaz de

armazenar a fórmula tratada. Assim sendo, a redução do espaço necessário para o armazenamento interno de uma fórmula lógica é uma qualidade que deve ser buscada pelos algoritmos resolvedores. Uma outra vantagem de evitar a repetição desnecessária de trechos internos da fórmula, é que decisões capazes de solucionar a sub fórmula não duplicada serão automaticamente propagadas por todas as ocorrências da sub fórmula.

Evitar repetição previamente conhecida é simples, basta que todos os operadores que operem sobre a sub fórmula tenham uma aresta partindo deles e incidindo sobre um único nó, que representa o operador raiz da sub fórmula em questão. Porém, dessa maneira, é possível que um nó pai tenha uma profundidade menor que um nó filho. A estrutura de dados que permite esse comportamento, é definida como um grafo e não mais uma árvore. Ou seja, a fórmula não clausal deve agora ser representada por um grafo com arestas direcionadas, ou, um grafo direcionado.

Entretanto, a utilização de um grafo direcionado, aceita a presença de ciclos. Um ciclo é representado por um caminho que, percorrendo as arestas da estrutura, consegue retornar a um nó já visitado no caminho. Se, sintaticamente, isso é possível, semanticamente, tais estruturas tornam impossível definir a fórmula lógica representada. Assim sendo, tais ciclos devem ser evitados. Para isso, é necessário utilizar uma variante da estrutura do grafo direcionado, conhecida como grafo direcionado acíclico, ou DAG, na sigla em inglês.

Baseado no mesmo princípio, as váriaveis proposicionais da fórmula também devem ser representadas uma única vez na estrutura final. Além da redução do tamanho total da estrutura, com essa medida todas as incidências de tal váriavel ficam centralizadas em um único ponto. Porém, três decisões são possíveis no que se refere à representação dos literais negados referentes a váriavel em questão:

- Um nó, representando o operador de negação, pode ser inserido na estrutura como pai do nó que representa a proposição. A incidência das arestas sobre essa proposição, nesse caso, será dividida entre ambos, respectivamente.

- Outra opção é a criação de dois nós, um representando o literal positivo e outro representando o literal negativo dentro da estrutura.

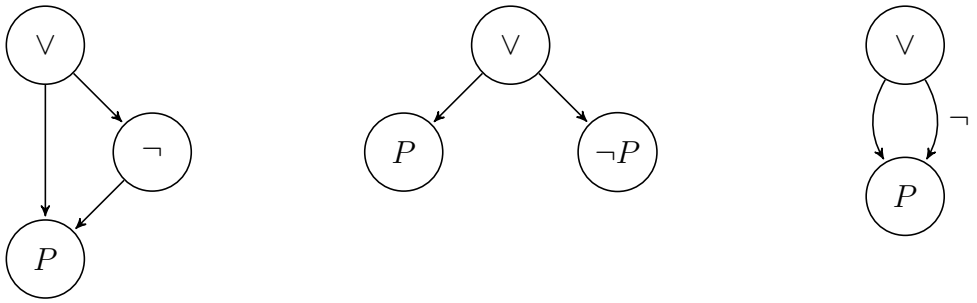

Figura 5.3: Formas de Representação para Sub-Fórmulas

- Uma terceira opção é a inserção de mais um tipo de aresta. Nesse caso, um conjunto de arestas representando as ligações com o nodo positivo e outro conjunto representando as ligações com os literais complementares.

Na figura 5.3, exemplos das três representações apresentadas para a sub-fórmula $(P \vee \neg P)$.

A segunda opção, referente a inserção de um segundo nó representando o literal negado, dificulta o processo de decisão da proposição P. Dois nós precisam ser visitados a cada decisão tomada. Já a terceira opção, apesar de ser a mais econômica em relação ao consumo de memória, aumenta a complexidade do processo de navegação pela estrutura da fórmula. No caso da primeira opção, seu formato permite que a negação esteja presente em qualquer ponto da fórmula, e não apenas nos literais. Isso permite que um número maior de fórmulas lógicas proposicionais seja tratado pelo algoritmo. Por essa razão, a primeira opção foi a escolhida, neste trabalho, como a forma de representar o operador de negação na fórmula.

A fórmula lógica proposicional apresentada anteriormente, pode ser representada pelo grafo direcionado acíclico, da figura 5.3:

Visando facilitar a utilização de um grafo direcionado como forma de representação de uma fórmula lógica proposicional por um algoritmo resolvedor do problema de satisfatibilidade, as seguintes características devem ser adotadas:

- Cada nó da estrutura deve armazenar informações sobre o elemento que representa.

- Os dois dados básicos que devem ser armazenados são o tipo do elemento e o iden-

$$((P \wedge Q) \vee \neg R \vee \neg S) \wedge (P \wedge Q)$$

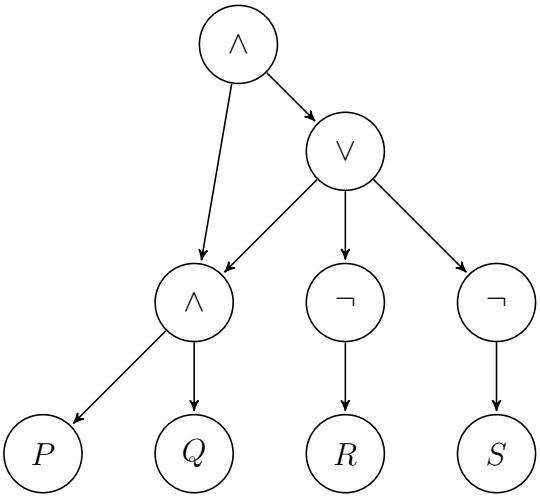

Figura 5.4: Representação de uma Fórmula Lógica Proposicional em um Grafo Direcionado Acíclico

tificador do mesmo.

- O tipo deve representar se o nó representa uma váriavel ou um operador, e no caso de ser um operador, qual.

- Já o identificador, que deve ser único em toda a fórmula, pode guardar o valor definido pelo formato ISCAS.

- Costuma se utilizar números inteiros para o identificador de um elemento. Para uma váriavel proposicional, esse valor pode ser positivo para literal positivo e negativo para o literal complementar.

Para facilitar a navegação, a partir de qualquer nó da estrutura, cada um deve armazenar duas listas: a lista do seus nós filhos e a lista do seus nós pais. Essas listas, tecnicamente, podem ser representadas por listas encadeadas ou por vetores de ponteiros para outros elementos da estrutura. A escolha deve ser tomada com base na necessidade, ou não, de inserção de novos elementos nessas listas durante o processo. Dessa forma toda e qualquer aresta da fórmula se torna bi-direcional e, assim sendo, se torna possível percorrer a fórmula partindo do seu nó raiz, e um nó folha ou de qualquer outro nó da estrutura.

5.4 Estruturas Internas Complementares

Nas seções anteriores, foram descritos os padrões e as estruturas necessários para a representação e o armazenamento de fórmulas não clausais no contexto de um algoritmo resolvedor para o problema de satisfatibilidade. Apesar de toda informação armazenada até o momento, estruturas adicionais devem ser utilizadas para facilitar as tarefas do algoritmo. Essas estruturas vão desde as mais genéricas, utilizadas várias vezes no decorrer da solução, até estruturas específicas que visam facilitar o armazenamento e o acesso de informações complementares, geralmente manipuladas nos passos intermediarios do resolvedor.

Entre as estruturas genéricas, três se destacam por serem mais utilizadas em algoritmos para esse fim específico: a lista encadeada, a pilha e a fila. A implementação de versões genéricas dessas estruturas,permite uma padronização no código do algoritmo, facilitam a manutenção e a resolução de problemas e permitem, dependendo da implementação e da situação, substituições de uma por outra. Essa possibilidade da simples substiuição de uma estrutura de pilha para uma de fila, por exemplo, permite que várias técnicas diferentes sejam testadas na mesma solução. Um exemplo disso foi descrito anteriormente, quando tratamos da forma de varredura, se em profundidade ou largura, da fórmula inicial representada no formato ISCAS.

Entre as várias informações necessárias para o correto funcionamento de um algoritmo resolvedor, a principal delas é aquela que permite um acesso direto a qualquer elemento que compõem a fórmula. A remoção do tempo necessário para percorrer a estrutura do grafo, em busca de um elemento específico, permite um melhor desempenho ao algoritmo resolvedor. Tal estrutura pode ser construída a partir de uma lista. Cada posição da lista apontará para um único elemento, operador ou váriavel, no grafo fórmula.

A implementação desta lista, e de outras presentes no algoritmo, pode utilizar duas formas de implementação, vetores ou listas encadeadas. A utilização de vetores permite um acesso indexado direto, enquanto o acesso através de uma lista encadeada, mesmo que ordenada por algum indexador, exige algum processo de busca dentro da mesma. Porém, a inserção e remoção de elementos da lista, além dos limites estabelecidos na sua criação,

é facilitada. Já a expansão do tamanho de um vetor se mostra mais complexa, exigindo a realocação do espaço de memória incialmente definido. Dessa maneira, a velocidade necessária para o acesso aos dados da lista e a possível necessidade de expansão do seu tamanho inicial, devem ser levados em conta no instante de decidir sobre qual técnica utilizar.

Visando permitir o acesso direto, o identificador único de cada nó do grafo pode ser utilizado como indexador da lista de todos os elementos. Dessa maneira, é possível acessar qualquer nó do grafo que representa a fórmula internamente, apenas com o seu identificador único.

Outras listas podem ser criadas, dependendo de que tipo de informação é necessária para a heurística analisada no resolvedor. Uma lista com todos os elementos que ainda não foram decididos, uma lista com as proposições que mais causaram conflitos, uma lista, ordenada com os operadores que possuem mais filhos, são exemplos de dados que podem ser armazenados, dependendo da técnica aplicada na solução. Pilhas e filas podem ser usadas de forma semelhante durante um processo de busca, por exemplo.

A plataforma deve fornecer abstrações das três estruturas principais, para facilitar o processo de análise de uma nova heurística proposta. Além disso, outras estruturas necessárias para o processo do resolvedor estão presentes. O grafo de implicação e a fórmula FNC, responsável por armazenar as cláusulas aprendidas, são exemplos dessas estruturas.

5.5 Processo Básico do Resolvedor

Após o processo de leitura e armazenamento da fórmula lógica proposicional, o algoritmo resolvedor já é capaz de iniciar o processo que busca decidir se uma fórmula é ou não satisfatível. Baseado nos principais algoritmos conhecidos, a prática mais comum, e que tem apresentado os melhores resultados, é a aplicação dos passos definidos pelo resolvedor DPLL[4]. Esses passos consistem na aplicação de três regras básicas. As regras do literal puro e da cláusula unitária, consideradas regras de implicação, devem ser utilizadas sempre que possível. Sempre que nenhuma das duas regras puderem ser aplicadas, uma

proposição deve ser escolhida e a regra da ramificação ocorre. O processo de aprendizado e retrocesso deve ocorrer sempre que um conflito for detectado.

A primeira etapa do processo, aplicação da regra do literal puro é de simples execução pelo resolvedor apresentado. A localização de um literal puro consiste em executar uma busca por todos os nós folhas do grafo. Um literal puro será aquele que não possuir nenhum operador de negação como pai. Isso ocorre pois, para cada proposição da fórmula, um único nodo é armazenado. Com a lista de todos os literais puros da fórmula, o resolvedor deve atribuir um valor-verdade para a proposição que ele represente, de tal maneira que o literal se torne verdadeiro. O valor verdade atribuído, pode ser armazenado em uma estrutura semelhante à lista de elementos, mencionada anteriormente. Para simplificar o acesso, toda a informação referente a um elemento da fórmula é armazenada em uma tabela de símbolos. Essa tabela, também indexada pelo identificador, deve guardar informações como valor-verdade atual e nó do grafo que representa o elemento.

Outra maneira de localizar os literais puros de uma fórmula que só possua operadores de negação junto aos literais, é percorrendo todos os nós que representam um operador de negação. A cada nó, deve se verificar sobre quais literais ele opera. Esses literais devem ser removidos da lista de possíveis literais puros, que possui todos os literais da fórmula inicialmente. Tanto nessa técnica quanto na apresentada anteriormente, uma estrutura auxiliar de lista, contendo todos os nós que representam uma negação ou todos os nós folhas, pode facilitar o acesso, melhorando o desempenho do algoritmo.

A atribuição de um valor-verdade a um literal puro, no estágio inicial do processo, permite que esse literal seja removido da fórmula. Essa remoção só é possível pois essa decisão não necessita de revisão em nenhuma etapa futura do algoritmo. O processo de remoção é realizado com a substiuição do nó que representa o literal, por um nó que represente a proposição Verdadeira ou Falsa, dependendo do valor-verdade atribuído.

A presença de um nó representando a proposição Verdadeira, ou Falsa, permite que simplificações ocorram na fórmula. Se um nó, representando uma disjunção, possui a proposição Verdadeira como filho, ele pode ser considerado verdadeiro. O nó da disjunção verdadeira deve ser substituído pelo nó que representa a proposição Verdadeira, e as

arestas que ligavam ele aos seus filhos devem ser removidas. Todo nó, com exceção do nó raiz da fórmula, que não possuam arestas incedindo sobre, não são mais alcançáveis a partir da raiz da fórmula. E, por isso, devem ser removidos do grafo. A simplificação também ocorre quando um nó de conjunção tem a proposição Falsa como filho. O processo de simplificação deve ser exceutado sempre que possível.

O processo de simplificação também funciona no sentido dos pais para os filhos. Caso uma conjunção seja verdadeira, todos os seus filhos devem ser verdadeiros tambémr, e a propagação do nodo representante da proposião verdadeira também chega a eles. Analogamente, uma disjunção falsa implica que todos os disjuntos devem ser falsos também, e o processo de simplificação deve prosseguir com esses nós filhos.

As atribuições realizadas pela regra de ramificação não podem executar as mesmas simplificações da regra do literal puro. O processo de remoção dos nós de um grafo é díficil de ser desfeito, e atribuições realizadas pela regra de ramificação precisam ser revistas com frequência. Porém, implicações como a decisão de uma disjunção ser verdadeira, pela presença de um filho verdadeiro, ainda são possíveis. Ao invés da remoção de nós do grafo, esses nós devem ter um valor atribuído. Esse valor é normalmente armazenado na tabela de símbolos. Dessa forma, no algoritmo proposto, operadores lógicos também podem ter um valor verdade atribuído.

Sempre que um valor é atribuído a qualquer elemento da fórmula, um valor referente ao nível de decisão deve ser armazenado. Esse valor, inicialmente igual a 0, deve ser incrementado toda vez que uma execução da regra de ramificação ocorrer e propagado para todas as implicações possíveis. Dessa maneira, quando uma decisão precisar ser revista, o algoritmo poderá percorrer todos os elementos com o mesmo nível de decisão da valoração revista, e desfazer a atribuição. Esse conceito permite o retrocesso de qualquer atribuição que não elimine nodos do grafo.

A regra da cláusula unitária, fortemente atrelada ao conceito de cláusula das fórmulas na FNC, possui um equivalente nas fórmulas não clausais. Sempre que uma disjunção, decidida como verdadeira por alguma atribuição anterior, possuir todos os seus filhos valorados como falsos, há exceção de um único elemento ainda não decidido, esse filho deve

ser valorado como verdadeiro. Isso ocorre, pois, caso o elemento seja valorado como falso, um conflito foi criado, com uma disjunção verdadeira sendo que todos os seus disjuntos são falsos. O mesmo processo ocorre para uma conjunção verdadeira onde todos os filhos, menos um, são verdadeiros. O filho não valorado deve ser atribuído como verdadeiro também.

Para que o algoritmo possa encontrar com facilidade operadores capazes de realizar as implicaçãoes descritas, o mesmo conceito de monitoria de literais, apresentado pelo algoritmo SATO, pode ser utilizado. Basta que cada nó guarde dois ponteiros, responsáveis por monitorar dois de seus filhos. Cada vez que um filho monitorado, de uma disjunção verdadeira, se torna falso, outro filho, não decidido, deve ser monitorado em seu lugar. Se, em algum momento, apenas um filho pode ser monitorado, uma implicação é possível. No caso de nenhum filho atender aos requisitos, um conflito foi detectado. O monitoramento segue padrões semelhantes para os diversos casos de implicações.

Visando facilitar o processo de verificação do valor-verdade de um filho, é importante que cada nó do grafo tenha um apontador para a sua posição específica na tabela de símbolos. Essa conexão de volta é útil quando o nodo é alcançado através de uma navegação pelo grafo. O acesso direto evita que uma busca na tabela pelo indexador seja necessária.

O procedimento repete os passos de atribuição e propagação de implicações enquanto um conflito, ou um modelo válido, não for encontrado. Um conflito ocorre quando existe a tentativa de valorar um nodo, já valorado, com seu valor-verdade complementar. Nesse caso o processo de retrocesso deve ser executado. É possível que o processo de valoração e retrocesso se repita inúmeras vezes, especialmente se nenhuma técnica de aprendizado for utilizada. Um modelo é encontrado quando o nó raiz é atribuído como verdadeiro. Essa é uma condição necessária, porém, não suficiente. Mesmo que ela ocorra, se após todas as implicações existirem na fórmula conjunções verdadeiras onde algum filho ainda não foi decidido, conjunções falsas onde nenhum filho foi decidido como falso, disjunções verdadeiras onde nenhum filho verdadeiro foi encontrado ou, ainda, disjunções falsas com filhos ainda não valorados, o procedimento deve continuar. Uma estrutura auxiliar, ar-

mazenando todos os operadores que estejam nessa situação, é útil nesse momento. Tal situação pode ocorrer pois, dependendo da técnica empregada, qualquer elemento da fórmula pode ter seu valor-verdade decidido em um dado momento e a propagação desse valor deve ocorrer sempre que possível. Um simples exemplo pode ser demonstrado com a seguinte fórmula lógica proposicional, $(P \vee (Q \wedge \neg Q))$. Se um resolvedor, baseado em uma heurística qualquer, atribuir ao elemento $\wedge$ dessa fórmula, verdadeiro, esse valor pode ser propagado ao elemento pai, $\vee$, visto que se ele possui um filho verdadeiro, também será verdadeiro. Porém, nada pode se dizer com certeza dos filhos de $\wedge$, seus valores-verdade só serão decididos em etapas futuras do resolvedor. Nesse momento, a raiz da fórmula será verdadeira, porém, a fórmula só será satisfatível, se Q e $\neg Q$ forem verdadeiros, e eles devem ser armazenados em uma estrutura auxiliar. Como tal situação não é possível, a decisão de atribuir verdadeiro ao elemento $\wedge$ se mostrou equivocada, já que não existe uma interpretação onde os seus filhos serão ambos verdadeiros.

O processo de execução do resolvedor base apresentado aqui pode utilizar as vantagens da computação paralela. Porém, as informações referentes aos nós não poderão mais ficar armazenadas no próprio grafo. Dados como valores atribuídos e fihos monitorados, devem ser armazenados exclusivamente na tabela de símbolos. O grafo deve ficar centralizado para as várias instâncias do resolvedor. É possível adicionarmos à estrutura auxiliar, com todos os operandos que devem ser decididos antes que um modelo seja encontrado, mencionada anteriormente, à tabela de símbolos. Com isso, basta que cada instância possua sua própria tabela de símbolos. Uma instância central deve ser reservada para gerenciar conflitos encontrados ou uma solução encontrada. As estruturas de aprendizado presentes também devem ser centralizadas.

5.6 Adicionando Técnicas de Aprendizado à Plataforma

A principal característica dos algoritmos resolvedores que apresentam as melhores performances é possuir uma boa técnica de aprendizado. Até esse momento, foi descrito um possível resolvedor que, apesar de funcional, executará o processo de busca com base em um método semelhante ao da força bruta. Nesse caso o resolvedor testará um número

muito grande de valorações antes de poder chegar a alguma conclusão, no pior caso, todas as valorações, presentes na tabela verdade da fórmula, serão testadas. Entre tantas valorações testadas sem sucesso, muita informação pode ser armazenada e utilizada para guiar o processo de busca.

Em capítulos anteriores, foi apresentado o processo de busca introduzido pelo algoritmo resolvedor Chaff. Esse processo consiste em, com o auxílio de um grafo de implicações, resumir as possíveis decisões, responsáveis pelo insucesso de uma valoração, a uma cláusula aprendida. Esse cláusula pode ser armazenada em uma fórmula na FNC e verificada, a cada nova valoração, para evitar que os mesmos erros tornem a acontecer. O grafo utilizado para obter essas informações é semelhante ao DAG utilizado para representar a fórmula, logo, já está presente na plataforma. Outra informação relevante é o nível de decisão de cada atribuição, também já presente na estrutura da fórmula, sendo utilizado no retrocesso após comflitos.

As claúsulas aprendidas devem ser armazenadas em uma lista de cláusulas, que por sua vez são uma lista de literais. A plataforma deve fornecer um método de, a cada alteração na valoração atual, verificar se a mesma não gera alguma claúsula vazia, entre todas as aprendidas. Caso isso ocorra, essa atribuição irá gerar um conflito na fórmula. Uma máquina, reponsável por verificar a existência de cláusulas unitárias entre as cláusulas aprendidas, também pode ser utilizada para a geração de novas implicações. O tempo de armazenamento de uma cláusula aprendida e a frequência na verificação dessas cláusulas deve ficar a critério do resolvedor analisado.

A plataforma deve fornecer métodos que facilitem a localização dos diversos UIPs no grafo de implicação. Dessa forma o resolvedor analisado pode verificar qual a melhor técnica de criação de cláusulas advindas de conflitos.

No próximo capítulo será apresentado e analisado um resolvedor simples, baseado na plataforma apresentada. As implementações, referentes as várias questões levantadas nesse capítulo, serão apresentada.

CAPÍTULO 6

IMPLEMENTAÇÃO DA PLATAFORMA

No capítulo anterior, foram discutidos detalhes sobre a plataforma para comparação de heurísticas para SAT não clausal apresentada nesse trabalho. Porém, com um enfoque mais conceitual sobre as principais etapas do algoritmo resolvedor. Nesse capítulo, os principais pontos serão revisitados com uma análise técnica sobre a implementação adotada em cada um deles. Serão analisadas as principais decisões de implementação adotadas, apresentando, sempre que relevante, o código-fonte correspondente.

Como apresentado anteriormente, as fórmulas lógicas não clausais são representadas utilizando o formato ISCAS. Através dessa estrutura é possível encontrar todas as variáveis lógicas da fórmula, assim como o operador raiz da mesma. Esses elementos podem ser localizados de forma direta através das entradas definidas como "INPUT" e "OUTPUT". Para a leitura dos demais operadores, o resolvedor deve percorrer toda a fórmula e armazená-los durante o processo. Como cada operador define seus operandos através do nome do mesmo, só após o resolvedor conhecer todos é que será capaz de montar a estrutura da fórmula.

Na implementação proposta para a plataforma estudada, uma função definida como "iscasparser" receberá o nome de um arquivo contendo uma fórmula lógica, válida, no formato ISCAS e será responsável pela leitura e criação das estruturas necessárias. Para tal, a função deverá percorrer, sequencialmente, cada um dos *tokens* da fórmula. Um tolkien é definido como qualquer operador ou variável lógica da fórmula. Para cada um desses *tokens*, um nó deve ser adicionado dentro de uma estrutura de dados própria. O trecho de código apresentado no algoritmo 6.1 mostra de forma simplificada a função "iscasparser".

Algoritmo 6.1: Função iscasparser

```
1 iscas iscasparser(char *filename)
```

```
{
    while( (token=gettoken(file)))
    {
        if(!strncmp(token,"INPUT",sizeof("INPUT")))
            last=INPUT;

        else if(!strncmp(token,"OUTPUT",sizeof("OUTPUT")))
            last=OUTPUT;

        else if(!strncmp(token,"AND",sizeof("AND")))
        {
            last=CLAUSE;
            iscasmap[lastclause]->type=AND;

            map<const char*, iscas_ds*,ltstr>::iterator cur;
            cur=iscasmap.find(lastclause);
            vector_pushback(&clauselist,&clausecount,&clauselistmax,
                    (cur->second));
        }
        else if(!strncmp(token,"OR",sizeof("OR")))
        {
            last=CLAUSE;
            iscasmap[lastclause]->type=OR;

            map<const char*, iscas_ds*,ltstr>::iterator cur;
            cur=iscasmap.find(lastclause);
            vector_pushback(&clauselist,&clausecount,&clauselistmax,
                    (cur->second));
        }
        else if(!strncmp(token,"NOT",sizeof("OR")))
        {
            last=CLAUSE;
            iscasmap[lastclause]->type=NOT;
        }
        else if(!strncmp(token,")",sizeof(")")))
```

```
37          {
38              last=NONE;
39          }
40          else if(last==INPUT)
41          {
42              map<const char*, iscas_ds*,ltstr>::iterator cur;
43              iscasmap[token]=initialize(TERMINAL,token);
44              cur=iscasmap.find(token);
45              cur->second->code = inputcount;
46              vector_pushback(&inputlist,&inputcount,&inputlistmax,
47                      (cur->second));
48              last=NONE;
49          }
50          else if(last==OUTPUT)
51          {
52              last=NONE;
53              output=initialize(CLAUSE,token);
54              iscasmap[token]=output;
55          }
56          else if(last==CLAUSE)
57          {
58              last=CLAUSE;
59              map<const char*, iscas_ds*,ltstr>::iterator cur;
60              map<const char*, iscas_ds*,ltstr>::iterator tmp;
61              tmp=iscasmap.find(lastclause);
62              cur=iscasmap.find(token);
63              assert(cur != iscasmap.end());
64
65              if(iscasmap[token]->type==NOT)
66              {
67                  cur=iscasmap.find(iscasmap[token]->son[0]->id);
68                  assert(cur != iscasmap.end());
69                  vector_pushback(&(iscasmap[lastclause]->negson),
70                          &(iscasmap[lastclause]->nonegson),
71                          &(iscasmap[lastclause]->__negsonmax),
```

```
72                          (cur->second));
73                  /*Create reverse*/
74                  if(iscasmap[lastclause]->type != NOT)
75                  {
76                      vector_pushback(&(cur->second->negparent),
77                              &(cur->second->nonegparent),
78                              &(cur->second->__negparentmax),
79                              (tmp->second));
80                  }
81              }
82              else
83              {
84                  vector_pushback(&(iscasmap[lastclause]->son),
85                          &(iscasmap[lastclause]->noson),
86                          &(iscasmap[lastclause]->__sonmax),
87                          (cur->second));
88                  /*Create reverse*/
89                  if(iscasmap[lastclause]->type != NOT)
90                  {
91                      vector_pushback(&(cur->second->parent),
92                              &(cur->second->noparent),
93                              &(cur->second->__parentmax),
94                              (tmp->second));
95                  }
96              }
97              if(iscasmap[token]->type == TERMINAL)
98                  iscasmap[lastclause]->noinput++;
99          }
100     }
101 }
```

Uma outra maneira de recriar a fórmula lógica representada pelo arquivo ISCAS seria, partindo do operador definido como raiz, buscar os operadores sobre os quais ele opera e adicioná-los na estrutura. Esse processo continuaria de maneira recursiva até que todos

os operandos fossem encontrados. Apesar dessa estratégia gerar uma implementação mais simples, o tempo de processamento necessário seria muito elevado para tratar problemas reais, visto que para cada operando uma nova busca pela fórmula seria necessária.

Cada nó inserido na estrutura possui três listas que são preenchidas durante o processo. A primeira contém todos os nós que estão acima deste na hierarquia da fórmula, mas que operam sobre o mesmo. Esses operadores são denominados pais do nó em questão. As outras duas listas contêm os nós filhos do operador em questão. Ou seja, aqueles nós sobre os quais o novo operando agirá. Os filhos são divididos em duas listas dependendo se são nós negados ou não. Um filho negado é aquele que na fórmula final aparece com um operador de negação logo abaixo do operando. Essa distinção entre os filhos facilita o tratamento do resolvedor que, dependendo da situação, precisa aplicar procedimentos distintos quando uma negação é encontrada. Além das tabelas, cada nó deve armazenar o seu tipo, se operador ou variável lógica, e o valor verdade a ele atribuído. No algoritmo 6.2, o código que define a estrutura de cada nó é apresentado.

Algoritmo 6.2: Estrutura de Dados de um Nó da fórmula

```
1  typedef struct __iscas_ds__ iscas_ds;
2
3  struct __iscas_ds__
4  {
5       int code;
6       char *id;
7       types type;
8       iscas_ds **son;
9       int noson;
10      int __sonmax;
11      iscas_ds **negson;
12      int nonegson;
13      int __negsonmax;
14
15      iscas_ds **parent;
16      int noparent;
17      int __parentmax;
```

```
18      iscas_ds **negparent;
19      int nonegparent;
20      int __negparentmax;
21
22      values value;
23 };
```

Como é possível verificar, cada uma das tabelas descritas anteriormente é, na verdade, uma lista de ponteiros para outros nós da fórmula. Durante o processo de criação dessa estrutura, toda vez que um elemento é adicionado como filho ou pai de um determinado operando, um ponteiro para o nó desse elemento deve ser adicionado a tabela correspondente. Porém, normalmente, um operando pode ainda não ter sido encontrado no momento em que um nó pai dele está sendo processado. Nesses casos, uma lista de elementos ainda não encontrados foi criada. Assim, sempre que um novo elemento da fórmula é adicionado a estrtura, a lista de operandos pendentes é verificada. Caso o novo elemento faça parte dessa lista, um apontador para o nó recém criado deve ser adicionado as tabelas necessárias.

Com todos os elementos devidamente representados pelos nós na estrutura e com cada uma das respectivas tabelas preenchidas, se torna possível a livre navegação pela fórmula. Partindo do nó representante do operador raiz, basta percorrer a tabela de filhos desse nó para descer um nível na fórmula lógica. O mesmo acontece partindo das variáveis lógicas, é possível subir um nível na estrutura da fórmula, apenas percorrendo a tabela de nós pais dessa variável. Executando de maneira repetitiva essas subidas e descidas, o resolvedor é capaz de percorrer todos os elementos da fórmula lógica.

Visando facilitar a navegação pela fórmula lógica, a plataforma também preenche uma nova estrutura. Essa estrutura contém um apontador para o nó que representa o operador raiz da fórmula, assim como uma tabela com apontadores para todos os operadores da fórmula. Uma outra tabela, com apontadores para todas as variáveis lógicas, também está presente. Por fim, dois contadores, um com o total de variáveis lógicas e outro com o total de operadores, também fazem parte da estrutura. Com essas informações, um

resolvedor possui acesso direto a qualquer elemento da fórmula e, pela própria estrutura do formato ISCAS, cada elemento será representado por um único nó. Com esse fato, o resolvedor pode decidir o valor verdade de qualquer elemento da fórmula e, através de um único ponto, expandir suas implicações lógicas. No algoritmo 6.3 a definição dessa estrutura mencionada.

Algoritmo 6.3: Estrutura Descritiva da Fórmula

```
1 typedef struct iscas
2 {
3       iscas_ds *output;
4       iscas_ds **input;
5       iscas_ds **clause;
6       int noinput;
7       int noclauses;
8  }iscas;
```

Após analisado o arquivo ISCAS e geradas todas as estruturas mencionadas, um resolvedor simples poderia ser criado com base no que foi visto. Uma fórmula lógica será considerada satisfatível se for possível encontrar um conjunto de valores verdades atríbuidos as variáveis lógicas que, após todas as implicações lógicas, tornem o operador raiz da fórmula vedadeiro. Partindo desse princípio, um resolvedor simples poderia definir como verdadeiro o operador raiz da fórmula expandindo suas implicações lógicas sucessivamente. Se o operador raiz for uma conjunção, todos os seus filhos, listados na estrutura, devem ser verdadeiros. Por outro lado, caso o operador raiz seja um disjunção, se ao menos um de seus filhos for verdadeiro será suficiente para que a decisão do nó raiz da fórmula como verdadeiro seja plausível. Dessa forma, é possível definir uma lista de nós que necessitam ser valorados como verdadeiro para que a valoração do nó raiz seja válida. Nesse caso, todos os filhos da conjunção devem ser adicionados a esta lista ou, no caso específico, ao menos um dos filhos da disjunção. Sequencialmente, cada um dos elementos deve ser removido dessa lista e ter o valor verdadeiro atribuído a ele. O processo de inserção dos seus filhos na lista ocorre da mesma maneira que aquele descrito para o operador raiz. O processo continua até que não exista mais elementos na lista, caso em que a fórmula

lógica se mostrou satisfatível e o valor verdade atribuído as variáveis lógicas é um modelo da fórmula. Na implementação da plataforma, o conceito de tal lista foi representado pela pilha apresentada no algoritmo 6.4.

Algoritmo 6.4: Pilha para armazenar os nós que devem ser decididos

```
1  struct stackPlace{
2      iscas_ds *node;
3      iscas_ds *reason;
4      struct stackPlace *prev;
5      struct stackPlace *next;
6  };
7
8  typedef struct stackPlace *STACKPLACEPTR;
9
10 struct stack{
11     STACKPLACEPTR top;
12 };
13
14 typedef struct stack *STACKPTR;
15
16 void push(STACKPTR stack, iscas_ds *node, iscas_ds *reason){
17     STACKPLACEPTR top, new;
18
19     new = newStackPoint();
20     new->node = node;
21     new->reason = reason;
22
23     top = stack->top;
24
25     new->prev = top;
26
27     if(top != NULL){
28         top->next = new;
29     }
30
```

```
31      stack->top = new;
32  }
33
34  iscas_ds* pop(STACKPTR stack){
35      iscas_ds *node;
36      STACKPLACEPTR top, newTop;
37
38      top = stack->top;
39      node = top->node;
40      newTop = top->prev;
41
42      if(newTop != NULL){
43          newTop->next = NULL;
44      }
45
46      stack->top = newTop;
47      free(top);
48      return(node);
49  }
```

Porém, caso um elemento que foi adicionado a lista não puder ter o valor verdadeiro atribuído, seja porque ele já possui o valor lógico falso ou porque algum de seus filhos não pode receber a atribuição lógica necessária, o resolvedor deverá tratar o problema. Nesse momento o processo principal deve ser parado e o processo de retrocesso deve ocorrer. Como a atribuição a esse elemento não foi possível, as atribuições feitas aos pais desse elemento também devem ser reprocessadas. Caso o pai em questão seja uma disjunção, algum outro nó filho, ainda não valorado, deve ser adicionado a lista de nós que devem ser decididos. Porém, caso todos os filhos já tenham sido decididos, ou o pai seja uma conjunção, a decisão atribuída ao nó pai deve ser desfeita e o processo de retrocesso continua com o próximo nó pai. Como um mesmo nó pode ter mais de um nó pai associado, um campo adicional foi inserido na estrutura da pilha. Esse campo define o nó pai, cuja atribuição resultou na inserção do nó na lista. No final, caso o

processo de alteração da distribuição lógica chegue ao nó raiz da fórmula, a mesma será dita insatisfatível. Uma implementação básica dessa estrutura é apresentada no algoritmo 6.5.

Algoritmo 6.5: Estrutura básica do resolvedor

```
1  push(stack, formula->output, NULL);
2
3  while (! isEmpty(stack)) {
4      node = pop(stack);
5
6      if (node->value != FALSE) {
7          if (node->type = AND) {
8              add_all_sons(stack, node);
9
10         } else if (node->type = OR) {
11             add_any_son(stack, node);
12         }
13     } else {
14         unsat = backtracking(stack, node);
15
16         if (unsat) {
17             break;
18         }
19     }
20 }
21
22 if (unsat) {
23     # Formula não é satisfatível
24
25 } else {
26     print_result(formula);
27
28 }
```

CAPÍTULO 7

CONCLUSÃO

Nos capítulos iniciais desse trabalho, apresentamos o problema de decidir sobre a satisfatibilidade de uma fórmula lógica proposicional. Após uma breve revisão dos conceitos por trás do problema, alguns dos seus vários casos de uso foram apresentados. O problema de satisfatibilida se apresenta como sub-problema de vários campos da computação. Dessa maneira, avanços tecnológicos nessa área refletem sobre muitas outras pesquisas. Essa é uma das principais características que mantêm o problema de satisfatibilidade como o foco central de grupos de pesquisas ao redor do mundo.

Como apresentado, muitos trabalhos se concentram sobre fórmulas em um formato bem específico, a Forma Normal Conjuntiva. As características técnicas desse formato possibilitaram o surgimento de técnicas que permitem aos resolvedores atuais tratarem problemas considerados extremamente complexos pelos sesus antecessores, em tempos muito reduzidos. O algoritmo DPLL aparece como peça central de um grande número de resolvedores.

Apesar de suas qualidades, o formato FNC se mostra excessivamente restritivo para a representação de alguns problemas, especialmente aqueles ligados a testes automatizados de circuitos. Visando aproveitar o grande conhecimento adquirido com o maquinário para FNC, muitas pesquisas tentam mapear entre os dois formatos as técnicas utilizadas. Porém, por se tratarem de classes diferenciadas de problemas, nem sempre as técnicas apresentam a mesma qualidade de desempenho. A escolha de quais dessas técnicas devem ser adotadas, não é uma tarefa trivial sem a realização de várias baterias de testes. E, ainda assim, mais de uma solulçao é possível.

Nesse contexto que o trabalho apresentado se insere. A proposta de criação de uma máquina resolvedora básica, utilizando-se das principais técnicas conhecidas, tenta facilitar o processo de desenvolvimento de um resolvedor para fórmulas não clausais. Um

conjunto de estruturas, otimizadas para o armazenamento de informações referentes ao problema também é apresentado. Este trabalho fornece uma base comum, útil para análises comparativas, entre as mais diversas heurísticas.

Por sua característica básica, muitos trabalhos podem utilizar as bases aqui apresentadas. Por se tratar de um resolvedor, otimizações sobre suas estruturas podem facilitar o desenvolvimento de novas técnicas. A utilizalçao de estruturas mais compactas de representação da fórmula adotada, a unificação do grafo que representa a fórmula, juntamente com o grafo de implicações, utilizado na ánalise de conflitos, são algumas das áreas que podem ser estudadas futuramente. Questões referentes a computação paralela, podem permitir ao algoritmo solucionar problemas intratáveis atualmente. Por fim, a implementação de técnicas específicas ao problema estudado, evitando o simples mapeamento, é uma das mais importantes pesquisas a serem realizadas.

BIBLIOGRAFIA

[1] DIMACS Challenge—Satisfiability: Suggested Format. 2009.

[2] David Bryan. The ISCAS '85 benchmark circuits and netlist format. 1985.

[3] J.M. Crawford e L.D. Auton. Experimental results on the crossover point in satisfiability problems. páginas 21–21, 1993.

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

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

[6] N. Eén e N. Sörensson. MiniSat: A SAT solver with conflict-clause minimization. *8th SAT*, 2005.

[7] H. Jain e E.M. Clarke. Efficient SAT solving for non-clausal formulas using DPLL, graphs, and watched cuts. páginas 563–568, 2009.

[8] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, e S. Malik. Chaff: Engineering an efficient SAT solver. páginas 530–535, 2001.

[9] Stuart J. Russell e Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[10] 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.

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

[12] J.P.M. Silva e K.A. Sakallah. GRASP: a new search algorithm for satisfiability. páginas 220–227, 1997.

[13] Z. Stachniak e A. Belov. Speeding-Up Non-clausal Local Search for Propositional Satisfiability with Clause Learning. *Lecture Notes in Computer Science*, 4996:257, 2008.

[14] C. Thiffault, F. Bacchus, e T. Walsh. Solving non-clausal formulas with DPLL search. *Lecture notes in computer science*, páginas 663–678, 2004.

[15] H. Zhang. SATO: An efficient propositional prover. *Lecture notes in computer science*, páginas 272–275, 1997.