Jacques Fux

# Análise de Algoritmos SAT para Resolução de Problemas Multivalorados

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

Belo Horizonte

8 de Outubro de 2004

# Folha de Avaliação

## Análise de Algoritmos SAT para Resolução de Problemas Multivalorados

Jacques Fux

Dissertação defendida e _____ **provada**, em 8 de Outubro de 2004 pela banca examinadora composta pelos professores:

---

Orientador *Ph.D* Claudionor José Nunes Coelho Junior
DCC - ICEX - UFMG

---

Professor *Doutor* Newton José Vieira
DCC - ICEX - UFMG

---

*Doutor* Luis Humberto Rezende Barbosa
DCC - ICEX - UFMG

## Resumo

O problema clássico $NP$-completo denominado SAT tem sido de muito interesse em diversas áreas da ciência da computação. Inúmeras heurísticas foram e são desenvolvidas com o intuito de atacar esse importante problema. Nessa dissertação faremos um estudo minucioso dos principais conceitos e das principais heurísticas para se resolver fórmulas booleanas SAT. A implementação e a apresentação de resultados empíricos se faz presente com o intuito de validar as melhores performances das heurísticas para diferentes classes de problemas. Apresentaremos também o conceito, os algoritmos e os resultados empíricos para heurísticas baseadas em lógica multivalorada o que representa um novo paradigma na forma de se trabalhar com o problema da Satisfabilidade.

## Abstract

The classical $NP$-complete problem of Boolean Satisfiability (SAT) has seen a lot of interest in several areas of Computer Science. A lot of heuristics have been developed to deal with this important problem. In this report, we define, analyze and present empirical results comparing the most important heuristics that deal with SAT and and check which better heuristic performances in distinct class of problem. We also explain the concept, the algorithms and the empirical results for heuristics based on multi-valued logic which represent a new paradigm dealing with Satisfiability problem.

*Aos meus queridos Papai, Mamãe e Benão*

# Agradecimentos

Gostaria primeiramente de agradecer ao Departamento de Ciência da Computação da UFMG por ter aberto uma oportunidade única para alguém de outra área. Agradecer também ao LECOM juntamente com o CNPq que me deram a oportunidade de receber uma bolsa de estudos.

Por certo esta dissertação não seria possível sem as contribuições do meu querido orientador Claudionor que desde a iniciação científica vem me ensinando muito a respeito da área acadêmica e da ciência da computação. Seus conhecimentos ilimitados, suas idéias geniais me fizeram admirar seu trabalho e desejar estudar cada vez mais para atingir um pouco mais do "saber".

Agradeço sinceramente aos meus amigos Ademir, Adriano e João Marcos que me ajudaram muito no tema desta dissertação dando sugestões brilhantes. À também aluna de mestrado Márcia pelas dúvidas e sugestões importantes e de última hora.

Meus agradecimentos ao professor Newton Vieira e ao Dr. Luis Humberto. Suas críticas e sugestões à minha dissertação me fizeram trabalhar com mais ardor.

Agradeço também ao professor Antônio Otávio que me ajudou nos momentos de aperto e sempre me recebeu com muito zelo.

Muitas discussões em torno do difícil assunto SAT foram travadas com o também aluno Romanelli Zuim. As trocas de informação contribuíram de forma positiva para a realização desta dissertação. Agradeço ao Zuim à paciência e amizade e desejo sorte em sua empreitada.

# Conteúdo

**Lista de Figuras** **vii**

**Lista de Tabelas** **ix**

**1 Introdução** **1**

1.1 Problemas $P$, $NP$ e $NPC$ . . . . . 1

1.2 O primeiro problema $NP$-completo . . . . . 2

1.3 Objetivos . . . . . 2

1.4 Contribuições da dissertação . . . . . 3

1.5 Estrutura da dissertação . . . . . 3

**2 SAT** **4**

2.1 O Problema SAT . . . . . 4

2.2 Aplicações de SAT . . . . . 5

2.3 Trabalhos Relacionados . . . . . 6

2.3.1 Comparação de Resolvedores SAT x BDDs . . . . . 7

2.4 Considerações Finais . . . . . 7

**3 Algoritmos para SAT** **8**

3.1 Introdução . . . . . 8

3.2 Davis-Putnam . . . . . 9

3.2.1 Exemplo . . . . . 11

3.3 Estruturas de Dados . . . . . 12

3.4 GRASP - *Generic Search Algorithm for Satisfiability Problem* . . . . . 16

3.4.1 Introdução . . . . . 17

3.4.2 Funções Importantes . . . . . 17

3.4.3 Estruturas do Processo de Procura . . . . . 18

3.4.4 Heurística GRASP . . . . . 19

3.4.5 Análise de Conflitos . . . 21
3.4.6 Exemplo . . . 22
3.5 SATO - *SAtisfiability Testing Optimized* . . . 23
3.5.1 Introdução . . . 23
3.5.2 Definições Básicas . . . 23
3.5.3 Principal Função . . . 24
3.5.4 Exemplo . . . 26
3.6 Zchaff . . . 28
3.6.1 Principais Funções . . . 29
3.6.2 Literais Observados . . . 30
3.6.3 Resolvente . . . 32
3.6.4 Exemplo . . . 32
3.7 Berkmin-Berkeley-Minsk . . . 33
3.7.1 Decisões de Berkmin . . . 34
3.7.2 Comparação de resultados . . . 35
3.8 Resultados Empíricos Preliminares . . . 36
3.8.1 *Graph Coloring Problem - GCP* . . . 36
3.8.2 *All Interval Series Problem - AIS* . . . 37
3.8.3 *Blocks World Planning Problem - BWP* . . . 38
3.8.4 *Logistics Planning Problem - LPP* . . . 38
3.8.5 *A Pipelined DLX Processor* . . . 39
3.8.6 *Pigeon Hole Problem - PH* . . . 39
3.8.7 *Morphed Graph Coloring Problem* . . . 40
3.9 Resultados Comparativos . . . 40
3.9.1 Análise de Resultados . . . 41
3.9.2 Conclusões . . . 43
3.10 Comparações - Resumo . . . 44
3.10.1 GRASP . . . 44
3.10.2 SATO . . . 45
3.10.3 Zchaff . . . 45
3.11 Considerações finais . . . 45

**4 Lógica Multivalorada** **46**
4.1 Introdução . . . 46
4.2 Definições . . . 46

4.3 Lógica Binária Estendida . . . . . 48
4.4 Lógica Ternária . . . . . 52
4.5 Lógica Quaternária . . . . . 53
4.6 Considerações finais . . . . . 54

**5 Algoritmos Revisitados** **55**
5.1 Introdução . . . . . 55
5.2 Davis-Putnam Binário Estendido . . . . . 55
5.2.1 Resultados . . . . . 56
5.3 Um resolvedor SAT para Lógica Multivalorada . . . . . 57
5.4 Análise de Conflitos de CAMA . . . . . 58
5.5 Resultados . . . . . 63
5.6 Considerações finais . . . . . 64

**6 Conclusões** **65**
6.1 Objetivos alcançados . . . . . 65
6.2 Conclusões . . . . . 65
6.3 Trabalhos Futuros . . . . . 66

**Bibliografia** **68**

# Lista de Figuras

2.1 Representação gráfica de vértices do BDD . . . . . 6
2.2 BDD para a fórmula $f = x_1 * (x_2 + x_3)$ . . . . . 7

3.1 Evolução dos Algoritmos . . . . . 9
3.2 Davis-Putnam [19] . . . . . 11
3.3 Exemplo da estrutura principal de Davis-Putnam . . . . . 12
3.4 Estruturas *lazy* [49] . . . . . 14
3.5 Grafo de Implicação I [45] . . . . . 19
3.6 Heurística GRASP [45] . . . . . 20
3.7 *Backtracking* não-cronológico [45] . . . . . 21
3.8 Exemplo da estrutura principal de GRASP . . . . . 23
3.9 Função *trie-merge* SATO [67] . . . . . 25
3.10 Exemplo da estrutura principal de SATO . . . . . 27
3.11 Exemplo *trie* SATO . . . . . 28
3.12 O algoritmo Zchaff [27] . . . . . 30
3.13 *Two Watched Literals* [42] . . . . . 31
3.14 Exemplo da estrutura principal de Zchaff . . . . . 33

4.1 Representação para $(1, X, 0)$ . . . . . 48
4.2 Representação *Double-rail* . . . . . 50
4.3 Representação de CNF em um circuito . . . . . 50
4.4 Resolvedor SAT para Lógica Binária Estendida . . . . . 51
4.5 Lógica Ternária . . . . . 52
4.6 Lógica Quaternária de Belnap . . . . . 54

5.1 Davis-Putnam Binário Estendido . . . . . 56
5.2 Grafo de Implicação II [40] . . . . . 58

5.3 Pseudo-Código MV-literais observados [40] . . . . . 60
5.4 Algoritmo para avaliar um literal multivalorado [40] . . . . . 60
5.5 Análise de Conflitos [40] . . . . . 61
5.6 Processo de aprendizado MV-SAT [40] . . . . . 62

# Lista de Tabelas

3.1 Comparação de resultados de SATO . . . . . 26
3.2 *Benchmarks* onde Berkmin se compara a Zchaff . . . . . 35
3.3 *Benchmarks* onde Berkmin é melhor que Zchaff . . . . . 35
3.4 *Graph Coloring Problem* . . . . . 36
3.5 *All-Interval Series - size 8* . . . . . 37
3.6 *All-Interval Series - size 10* . . . . . 37
3.7 *Blocks World Planning Problem - size 9* . . . . . 38
3.8 *Blocks World Planning Problem - size 11* . . . . . 38
3.9 *Logistics Planning Problem - size 5* . . . . . 39
3.10 *Logistics Planning Problem - size 13* . . . . . 39
3.11 *A Pipelined DLX Processor* . . . . . 39
3.12 *Pigeon Hole Problem* . . . . . 40
3.13 *"Morphed" Graph Colouring* . . . . . 40
3.14 Tabela dos Resultados *Latin Square* . . . . . 41
3.15 Tabela 1 dos Resultados de problemas industriais . . . . . 42
3.16 Tabela 2 dos Resultados de problemas industriais . . . . . 42
3.17 Tabela 1 dos Resultados de problemas especiais . . . . . 43
3.18 Tabela 2 dos Resultados de problemas especiais . . . . . 43
3.19 Comparação entre Heurísticas . . . . . 44

4.1 AND para Lógica Binária Estendida . . . . . 49
4.2 OR para Lógica Binária Estendida . . . . . 49
4.3 NOT para Lógica Binária Estendida . . . . . 49
4.4 Representação de Lógica Binária Estendida utilizando dois bits . . . . . 49
4.5 AND para codificação OHE . . . . . 51
4.6 OR para codificação OHE . . . . . 52

4.7 NOT para codificação OHE . . . . . 52
4.8 AND para Lógica Ternária . . . . . 53
4.9 OR para Lógica Ternária . . . . . 53
4.10 NOT para Lógica Ternária . . . . . 53

5.1 Resultados de um Multiplicador para Davis-Putnam Binário Estendido . . . . . 57
5.2 Resultados de uma ALU para Davis-Putnam Binário Estendido . . . . . 57
5.3 *Benchmarks* com duas diferentes heurísticas de decisões de CAMA . . . . . 63
5.4 Solução de CAMA sem utilizar técnicas multivaloradas para os *Benchmarks* . . . 64
5.5 Tabela Comparativa entre Zchaff e CAMA . . . . . 64

# Capítulo 1

# Introdução

*"O Coelho Branco pôs os seus óculos.*
*- Por onde devo começar, Sua Majestade? - perguntou.*
*- Comece pelo começo - disse o Rei, muito gravemente*
*- e continue até chegar ao fim: então pare."*
*(Lewis Carroll, Alice no País das Maravilhas, cap.XII)*

## 1.1 Problemas $P$, $NP$ e $NPC$

Nas primeiras décadas do século passado alguns matemáticos tentaram *formalizar* toda a matemática[1]. Em 1900 David Hilbert apresentou os grandes problemas da matemática com o objetivo de se resolver todos, já que não poderia haver algo que não pudéssemos conhecer. Infelizmente tal objetivo nunca foi alcançado. O famoso Problema da Parada de Turing mostrou que existem problemas que não podem ser resolvidos por nenhum computador [61, 62].

Algoritmos que, para entradas de tamanho $n$, possuem tempo de execução do pior caso é $O(n^k)$ são chamados **algoritmos de tempo polinomial**. Problemas que não podem ser resolvidos em tempo polinomial são chamados **algoritmos de tempo não-polinomial**. Definimos então os problemas **Tratáveis** como aqueles que podem ser resolvidos em tempo polinomial e os problemas **Intratáveis** como aqueles resolvidos em tempo superpolinomial [30, 60].

Entretanto existem problemas que ainda não sabemos se podem ser resolvidos utilizando um algoritmo polinomial. Chamamos essa classe interessante de problemas de $NP$**-completo**. A formulação $P \neq NP$ foi proposta em 1971 e constitui até hoje um dos grandes problemas dentro da teoria da computação e da Matemática [14].

[1] O problema de se axiomatizar toda matemática foi conhecido como o Projeto de Hilbert que tinha como objetivo mostrar que se a matemática fosse consistente e completa, no sentido em que com um número finito de postulados poderíamos saber de toda verdade matemática e jamais chegaríamos a uma auto-contradição. Gödel mostra que o preço de consistência é a eterna incompletude. Que não pode haver uma matemática complexa o suficiente, capaz de lidar com infinitos, sem que se desague necessariamente em paradoxos. Sempre haverá novos indecidíveis. Ou seja: se a estrutura lógica é grande o suficiente, ela produz necessariamente múltiplas verdades convivendo anárquica e consistentemente no sistema e quanto mais se avança em novos resultados, novos indecidíveis e novas múltiplas verdades aparecem. [28]

Apresentamos então três classes de problemas[2]. A classe $P$ consiste em problemas que podem ser resolvidos no tempo $O(n^k)$ para alguma constante $k$ e onde $n$ é o tamanho de entrada para o problema em questão. A classe $NP$ consiste em problemas *verificáveis* em tempo polinomial, ou seja, dada uma determinada solução (certificado) é possível verificá-la em tempo polinomial. Dadas as definições acima, percebemos claramente que $P \subseteq NP$. A grande questão é saber se $P$ é ou não um subconjunto próprio de $NP$ [60].

A outra classe de problemas particularmente interessante é a chamada $NPC$ ou $NP$-completos. Um problema pertence a essa classe se e somente se:

1. É um problema $NP$;
2. Qualquer outro problema $NP$ é redutível em tempo polinomial[3] a ele.

## 1.2 O primeiro problema $NP$-completo

Como citado no item anterior, para mostrarmos que um problema pertence à classe $NPC$ é necessário reduzir um problema reconhecidamente pertencente a $NPC$ a ele. Logo é necessário demonstrarmos a existência de um primeiro problema $NPC$ a partir do qual será possível construir a classe $NPC$. O problema utilizado é o da satisfabilidade de circuitos, no qual temos um circuito combinatório booleano composto por portas $AND$, $OR$ e $NOT$, e desejamos saber se existe algum conjunto de valores de entradas para esse circuito que faça sua saída ser 1.

A demonstração de que tal problema é da classe $NPC$ é discutida na refêrencia [30] onde os autores constroem, de maneira didática, a demonstração atribuída a Cook no artigo [13].

## 1.3 Objetivos

Até então fizemos uma breve explanação sobre as classes de problemas ($P$, $NP$ e $NPC$) e a existência de um primeiro problema pertencente à classe $NPC$ com o objetivo de situar e justificar o tema dessa dissertação. Devido ao fato de não se conhecer um algoritmo polinomial para a solução de problemas $NP$-completos, o foco dessa dissertação será relativo às **heurísticas**[4] para se encontrar soluções aproximadas. Apresentamos um texto didático mostrando as principais heurísticas e seus resultados para diversas classes de problemas. O estudo e a implementação das heurísticas reconhecidamente melhores para a solução do primeiro problema $NP$-completo bem como a análise de heurísticas para lógica multivalorada[5] serão o foco principal dessa dissertação.

[2] Existem inúmeras outras classes de problemas [30].

[3] Redução é a transformação de uma instância de um problema em uma instância equivalente de outro problema.

[4] Heurística é a arte e a ciência da descoberta ou da invenção. A palavra vem da raiz grega **eureka** que significa *procurar*. Uma heurística para um dado problema é uma forma de se achar uma solução apesar de nem sempre ser possível encontrá-la.

[5] Dizemos que a lógica Aristotélica é aquela com apenas dois valores. Semanticamente uma sentença pode assumir o valor *Verdade* ou valor *Falso*. Uma lógica que pode assumir outros valores é chamada de lógica multivalorada. A lógica *fuzzy* [12] é um exemplo bem conhecido dessa lógica.

## 1.4 Contribuições da dissertação

Em virtude de se trabalhar com a classe de problemas $NPC$, o estudo e a implementação de heurísticas torna-se fundamental. Nesta dissertação apresentamos as principais heurísticas que atacaram o problema de satisfabilidade, fazendo uma análise bem detalhada das suas principais potencialidades além de estender o foco à lógica multivalorada. A contribuição principal dessa dissertação é a de analisar as melhores heurísticas trabalhando com lógica binária e também propor e analisar heurísticas para lógica multivalorada.

A presente dissertação tem como objetivo fazer uma análise glogal das principais heurísticas binárias e multivaloradas como foi feita na referência [26]. Apresentamos também resultados empíricos importantes para diversas instâncias do problema de satisfabilidade. Em vários artigos, uma referência importante é [26]; porém percebemos a necessidade de melhorá-la devido a inúmeros trabalhos realizados nos últimos anos.

A metodologia utilizada nesta dissertação é através do estudo de casos. Muitos problemas foram escritos numa fórmula booleana e então resolvidos através das principais heurísticas implementadas. Utilizando tabelas comparativas, mostramos quais das heurísticas implementadas funcionam melhor para determinada classe de problema.

## 1.5 Estrutura da dissertação

O restante da dissertação é composta por 5 capítulos. No capítulo 2 definiremos formalmente o primeiro problema da classe $NP$-completo chamado Problema de Satisfabilidade (SAT) e mostraremos algumas pequenas aplicações. No capítulo 3 apresentaremos as principais heurísticas binárias em busca de uma boa solução para o problema SAT e apresentaremos resultados empíricos para diversas instâncias do problema (benchmarks [33]). Mostraremos qual heurística se comporta melhor e o porquê. No capítulo 4 defineremos formalmente lógica multivalorada mostrando duas formas de codificação (uma para lógica ternária e uma para lógica multivalorada). Apresentaremos as principais portas lógicas bem como a forma de se trabalhar com fórmulas normais conjuntivas multivaloradas. No capítulo 5 mostraremos duas heurísticas importantes para tratar problemas multivalorados apresentando resultados empíricos e, finalmente, no capítulo 6 apresentaremos nossas conclusões e projetos futuros desta dissertação.

# Capítulo 2

# SAT

*"The conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no ignorabimus." - Hilbert,1900*

## 2.1 O Problema SAT

Apresentamos agora o primeiro problema NP-completo. O problema de determinar se uma fórmula booleana é satisfeita ou não é provado $NP$-completo por Stephen Cook no artigo [13] Em [30] e [60] também são apresentadas demonstrações.

Uma expressão booleana pode ser contruída de [7]:

1. Variáveis que assumem valores booleanos, ou seja, assumem o valor 1(verdade) ou o valor 0 (falso).
2. Operadores binários $*$ and $+$ , referentes aos operadores lógicos $AND$ e $OR$.
3. Operador unário $\neg$ para a representação lógica da negação
4. Parênteses para grupo de operadores e operandos caso seja necessário alterar a precedência de operadores ($\neg$, $*$ e por último $+$).

Um exemplo de uma expressão booleana é $x * \neg(y + z)$. A subexpressão $(y + z)$ é verdade sempre que $y$ ou $z$ tem o valor 1, mas é falsa quando ambos $y$ e $z$ assumem o valor 0. A expressão maior $\neg(y + z)$ é 1 exatamente quando $(y + z)$ é falso, ou seja, quando ambos $y$ e $z$ assumirem o valor 0. Então se $y$ ou $z$ ou ambos assumirem o valor 1 então $\neg(y + z)$ é falso. Considerando a expressão inteira, temos um $AND$ de duas subexpressões que assume o valor 1 se ambas forem verdadeiras. Isso é verificado somente quando $x = 1$, $y = 0$ e $z = 0$. Logo estas atribuições satisfazem a expressão. Dizemos que uma expressão é satisfeita ou satisfatível se existe uma

atribuição ou assinalamento das variáveis que fazem com que uma fórmula normal conjuntiva (CNF)[1] assuma o valor 1.

O problema de satisfabilidade é o seguinte: Dada uma expressão booleana, ela é satisfeita?

Uma instância do problema de satisfabilidade (SAT) pode ser apresentada através de uma expressão booleana denominada fórmula normal conjuntiva (CNF). Uma fórmula booleana proposicional é constituída de literais e cláusulas. Como já explicado, cada variável assume valores pertencentes ao conjunto $\{0, 1\}$. Um literal então pode ser a própria variável $x$ ou seu complemento $\neg x$. Uma cláusula $c$ é definida como $c = (l_1 + l_2 + l_3 + ... + l_n)$ e pode ser definida pelo conjunto de literais $c = \{l_1, l_2, ..., l_n\}$. Uma CNF é a disjunção de cláusulas, ou seja, $f = (c_1 * c_2 * c_3 * ... * c_m)$ e pode também ser escrito como o conjunto de cláusulas $f = \{c_1, c_2, ..., c_m\}$ [27].

Já que estamos interessados em soluções para o problema SAT, algumas definições se farão úteis.

Seja $\varphi : \{x_1, ..., x_n\} \rightarrow \{0, 1\}$ um assinalamento para todas a variáveis booleanas $\{x_1, x_2, ...x_n\}$.

- Um literal $l$ é satisfeito se e somente se assume o valor verdade, ou seja, ($l \equiv x_i$ e $\varphi(x_i) = 1$) ou ($l \equiv \neg x_i$ e $\varphi(x_i) = 0$).
- Uma cláusula $c = (l_1 + l_2 + l_3 + ... + l_k)$ é satisfeita se e somente se pelo menos um literal $l \in c$ é satisfeito, ou seja, assume o valor 1. Uma cláusula vazia, aquela que não contém literais, denotada por $\emptyset$ é sempre não-satisfeita.
- Uma CNF $f = \{c_1, c_2, ..., c_m\}$ é satisfeita se e somente se todas as cláusulas assumem o valor 1.

Uma cláusula é dita **satisfeita** se pelo menos um dos literias assume o valor 1, **não-satisfeita** se todos os seus literais assumem o valor 0, **unitária** se todos os literais, exceto um, assumem o valor 0, e **não-resolvível** caso contrário (chamados **literais livres**[2]).

## 2.2 Aplicações de SAT

Aplicações em áreas diversas da computação tornam o problema SAT bastante estudado e trabalhado. A fórmula SAT é uma expressão poderosa e pode ser usada para definir diferentes restrições e relações. Em particular, pode ser usada para representar as relações inerentes a um circuito combinatório. Um mapeamento direto pode ser estabelecido de um circuito combinatório para uma fórmula CNF SAT. Um circuito combinatório pode ser expresso como uma *netlist*, com portas básicas e conexões ligando as portas. Cada ligação é representada por uma variável booleana na fórmula SAT. Cada porta é representada por uma cláusula na fórmula CNF.

SAT também pode ser usado para verificar a equivalência de dois circuitos combinatórios, similar à geração de teste onde dois circuitos alvo são combinados. O solucionador de SAT

[1] A definição formal de CNF se fará presente nos próximos parágrafos.

[2] São aqueles que ainda não assumiram nenhum valor, ou seja, não estão assinalados.

procura o caso em que as saídas dos dois circuitos são diferentes. Essa abordagem também permite testar a condição de equivalência adicionando as condições como se fossem cláusulas extra à fórmula [34].

Entretanto, nos concentraremos apenas na análise das principais heurísticas, tanto em lógica com dois valores quanto em lógica multivalorada, sempre atentos às diversas aplicações.

## 2.3 Trabalhos Relacionados

Existem diferentes abordagens para se trabalhar o problema da satisfabilidade de fórmulas booleanas. Uma forma interessante de se tentar tratar este problema é a estrutura de dados chamada *Binary Decision Diagram* ou simplesmente BDD.

Um BDD [9, 8] é um grafo acíclico dirigido que permite representar e manipular expressões booleanas. Através da utilização de BDDs é possível reduzir o espaço alocado para a representação das expressões em comparação com os métodos tradicionais, como árvores de decisão ou tabelas verdade.

BDDs são grafos acíclicos dirigidos formados por um conjunto $V$ de vértices. Os vértices de um BDD podem ser de dois tipos: terminais e não-terminais. Um BDD possui apenas dois vértices terminais, onde cada um contém um valor booleano. Então, sendo $v$ um vértice terminal, seu valor é denotado $valor$: $V \rightarrow \{0, 1\}$. Um vértice não-terminal $v$ é dado por $var(v) = x$ onde $x \in X$, onde $X$ é o conjunto variáveis de uma função booleana $f$. Os vértices não-terminais possuem pelo menos um arco de chegada e exatamente dois arcos de saída que são denotados por $zero(v)$ e $um(v)$, respectivamente. A Figura 2.1 mostra a representação gráfica dos vértices terminais e não-terminais para o $valor(v) = c \in \{0, 1\}$.

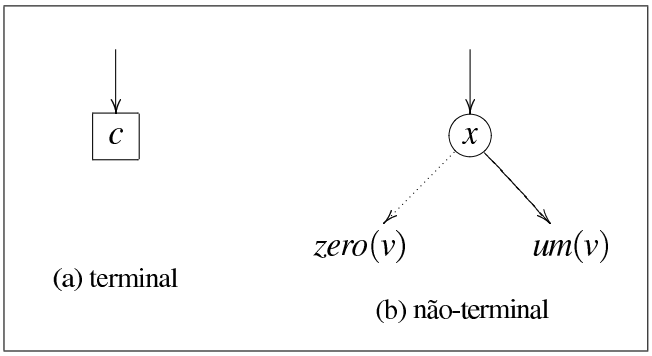


**Figura 2.1**: Representação gráfica de vértices do BDD

Apresentamos na Figura 2.2 um BDD para a fórmula $f = x_1 * (x_2 + x_3)$.

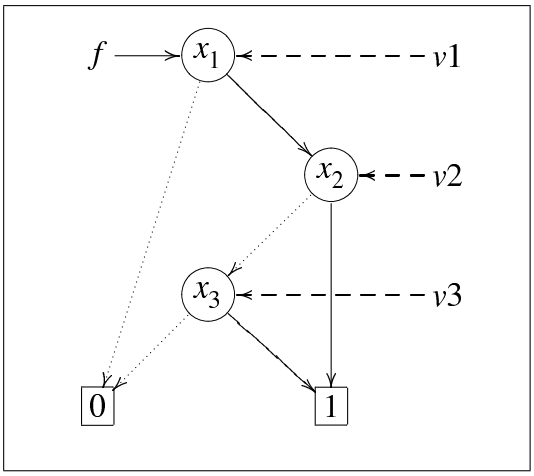


**Figura 2.2**: BDD para a fórmula $f = x_1 * (x_2 + x_3)$

Apesar de ser uma abordagem interessante, não entraremos em detalhes da resolução de problemas SAT utilizando BDDs. Entretanto podemos fazer uma breve comparação das vantagens e desvantagens de se resolver um problema SAT por meio de BDDs.

### 2.3.1 Comparação de Resolvedores SAT x BDDs

- BDDs funcionam em fórmulas arbitrárias enquanto que os resolvedores SAT funcionam apenas em fórmulas normais conjuntivas (CNFs).
- BDDs não conseguem resolver problemas da classe *Quasigroup* o que alguns resolvedores SAT fazem de forma simples. Porém, apesar dos resolvedores SAT abordarem os problemas da classe *Pigeonhole* [52], a solução via BDDs é mais direta em virtude das equivalências de fórmulas.
- Os resolvedores SAT são superiores na procura de uma solução simples no espaço de procura. Por outro lado não são úteis para achar equivalências de fórmulas booleanas, o que os BDDs fazem naturalmente.
- A capacidade de representação dos BDDs não atingiu níveis de modelagens reais necessários a problemas industriais, entretanto o seu tipo de representação permite que ingressem em ferramentas comerciais [9].

## 2.4 Considerações Finais

Neste Capítulo, também introdutório, fizemos as primeiras definições relativas ao problema SAT, os quais serão utilizadas ao longo desta dissertação. Apresentamos aplicações importantes e fizemos uma breve introdução aos BDDs.

# Capítulo 3

# Algoritmos para SAT

*Que caminho devo tomar? perguntou Alice ao Gato.*
*-Depende muito aonde você quer chegar.*
*-Qualquer lugar está bom.*
*-Então qualquer caminho que você escolher tá bom!*
*-Mas eu quero chegar a algum lugar.*
*-Ah, isso com certeza acontecerá se você andar o suficiente!*
*(Lewis Carroll, Alice no País das Maravilhas, cap.IV)*

## 3.1 Introdução

Dada uma fórmula proposicional, o cerne do problema SAT é verificar se existe uma atribuição de valores para as variáveis de uma equação tal que esta fórmula seja satisfeita [43].O problema SAT tem sido de grande interesse teórico e prático já que é um problema canônico $NP$-completo [13]. Problemas no campo da inteligência artificial [36] e testes de circuitos [48], podem ser formulados como instâncias do problema SAT, o que motiva a pesquisa em resolvedores SAT eficientes .

A procura de resolvedores SAT eficientes tem tido grandes sucessos. Os algoritmos SAT são baseados em vários princípios, como resolução [3], procura [19], procura local, entre outros [55]. Alguns destes algoritmos são ditos **completos** enquanto outros são ditos **estocásticos**. Para uma instância SAT, um algoritmo completo é aquele que acha uma solução ou prova que tal solução não existe. Métodos estocásticos[1], por outro lado, não provam que uma solução não existe. São utilizados em FPGAs[2] e problemas de Inteligência Artificial já que apenas a satisfabilidade é importante [50]. Em problemas de verificação a demonstração que um problema não tem solução é de fundamental importância e por isso são utilizados algoritmos completos.

Recentemente muitos algoritmos baseados na estrutura de dados Davis-Putnam Logemann-Loveland (DPLL) [19] estão surgindo como os mais eficientes métodos para algoritmos completos

[1]Métodos estocásticos foram utilizados antes do desenvolvimento de algoritmos completos e eram úteis para a procura de uma solução mais simples.
[2]Field-programmable gate array [31].

SAT. Nos últimos quarenta anos a pesquisa tem se voltado para algoritmos baseados no DPLL como NTAB [16], POSIT [22] CSAT [21] entre outros. Entretando os algoritmos mais eficientes e mais importantes, conceitualmente falando, são Sato [64], Grasp [46], Chaff [27] e Berkmin [24]. Na Figura 3.1 apresentamos a evolução desses algoritmos.

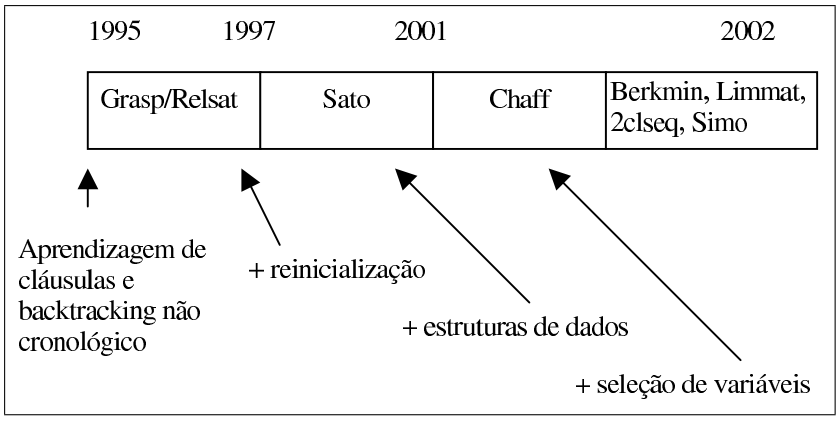


**Figura 3.1**: Evolução dos Algoritmos

Nas próximas seções explicaremos detalhadamente o funcionamento de tais algoritmos. Partiremos da formulação principal que é o método proposto por Davis-Putnam para podermos explicar melhor os outros algoritmos baseados nesse método.

## 3.2 Davis-Putnam

O algoritmo básico para solucionar SAT foi proposto por Martin Davis and Hillary Putnam, em 1960 [3]. Ao contrário de procurar enumerar cada combinação possível de valores para as variáveis, esse algoritmo utiliza técnicas de *backtrack* para reduzir o espaço de procura. A base do algoritmo é a seguinte: se um assinalamento parcial fizer uma cláusula ser avaliada para 0, esse assinalamento parcial não poderá ser parte de uma solução válida. O algoritmo procura identificar a não satisfabilidade em um assinalamento parcial de forma que não haja necessidade de tentar outras variáveis ainda sem valor atribuído. Isso normalmente resulta em uma procura mais eficiente que simplesmente enumerar cada uma das combinações possíveis de valores. Esse algoritmo é considerado completo, ou seja, se uma solução existe o algoritmo irá encontrá-la [19, 11, 10].

O algoritmo é recursivo e cria uma árvore de procura atribuindo valores e dividindo o problema em problemas menores. Utiliza uma técnica comum em heurísticas: dividir para conquistar. Empiricamente o algoritmo funciona bem na média, chegando a resultados satisfatórios para alguns tipos de problemas, porém seu pior caso, como esperado, é exponencial.

A idéia geral do algoritmo é:

1. Atribui-se 1 ou 0 para as variáveis que compõem pelo menos uma cláusula de uma única variável, de tal forma que essa cláusula seja 1 (verdadeira);
2. Deve-se verificar se não há conflito, ou seja, se o valor da variável que torna uma cláusula verdadeira não deixa outra cláusula com o valor 0 (falso);
3. Se ocorrer um conflito, o algoritmo deve ser interrompido, pois a fórmula não pode ser verdadeira;
4. Se não há conflito, as cláusulas com apenas uma variável que receberam o valor 1 (verdadeiro) são eliminadas da fórmula, pois, o resultado não depende mais do seu resultado;
5. As cláusulas com mais de uma variável que possuem aquela variável que recebeu o valor 1, podem ser eliminadas, caso não haja negação para essa variável com valor 1;
6. Quando não existirem mais cláusulas com apenas uma variável, escolhe-se uma variável qualquer e atribui um valor qualquer, por exemplo 1, para ela;
7. Após a substituição da variável escolhida por 1, deve-se verificar novamente se existem cláusulas com apenas uma variável e também se não ocorreu conflito;
8. Se um conflito for encontrado, antes de terminar o processamento do algoritmo com falha, deve-se retornar o processamento até o ponto que a variável escolhida recebeu o valor 1 e atribuir agora o valor 0, ou seja, o valor que ainda não foi processado;
9. Verifica-se novamente se existem cláusulas com apenas uma variável e também se não ocorreu conflito;
10. Se um conflito for encontrado com o segundo valor da variável escolhida, deve-se terminar o processamento do algoritmo com falha, pois não existe valor dessa variável que satisfaça à fórmula;
11. Realiza-se esse procedimento até que todas as cláusulas possam ser eliminadas;
12. Se não foi encontrada nenhuma variável que gere falha no processamento do algoritmo, pode-se afirmar que a fórmula pode ser satisfeita, ou melhor, pode ter um resultado verdadeiro (igual a 1).

Na Figura 3.2 apresentamos o pseudo-código do Davis-Putnam.

```
function Satisfiable (clause set S) return Boolean
    % unit propagation
    repeat
        for each (unit clause L in S) do {
                % unit subsumption
                (delete from S every clause containing L)
                (delete ¬L from every clause of S in which occurs) }
                   if (S is empty) then {
                return true }
         else if (a clause becomes null in S) then {
                return false }
     until (no further changes result)
    % splitting
    (choose a literal L occurring in S)
    if (Satisfiable ( S ⋃ {L} )) then {
          return true }
    else if (Satisfiable ( S ⋃ {¬L} )) then {
           return true }
         else {
            return false }
end function
```

**Figura 3.2**: Davis-Putnam [19]

### 3.2.1 Exemplo

Seja a seguinte fórmula normal conjuntiva (Figura 3.3):

$$(\neg x_1) * (x_1 + x_2 + x_4) * (x_3 + \neg x_1 + x_2)$$

Utilizando a regra de cláusulas unitárias[3], a heurística assume que $(\neg x_1)$ vale 1, chegando a fórmula:

$$(x_1 + x_2 + x_4) * (x_3 + \neg x_1 + x_2)$$

Ainda pela mesma regra, observamos que a cláusula $(x_3 + \neg x_1 + x_2)$ contém $(\neg x_1)$ e por isso eliminamos a cláusula ficando com:

$$(x_1 + x_2 + x_4)$$

Sabendo que $\neg(\neg x_1) = x_1$, eliminamos $x_1$ de $(x_1 + x_2 + x_4)$ chegando a fórmula:

$$(x_2 + x_4)$$

[3]Conforme descrito na idéia geral do algoritmo.

Assumindo $x_2 = 1$ temos então $(x_2 + x_4) * (x_2)^4$ recursivamente aplicando a regra da cláusula unitária, resolvemos o problema com o assinalmento $x_1 = 0$ e $x_2 = 1$. A fórmula é dita satisfeita.

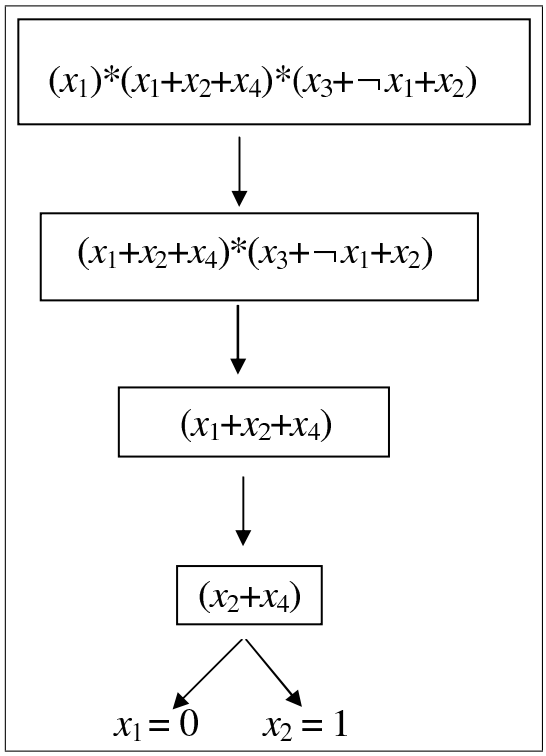


**Figura 3.3**: Exemplo da estrutura principal de Davis-Putnam

## 3.3 Estruturas de Dados

A grande maioria das heurísticas baseia-se na mesma estrutura de dados proposto no procedimento principal atribuido à Davis,Putnam e Lovelland.

Podemos encontrar dois tipos de estruturas de dados para os algoritmos baseados em DPLL: estruturas baseadas em contadores e estruturas de dados *lazy*. Ambas são capazes de mostrar de forma direta quando uma fórmula é satisfeita ou não, assim como identificar cláusulas unitárias. Para as estruturas de dados baseadas em contadores, as cláusulas são representadas como listas de literais e para cada variável mantém-se uma lista de todas as cláusulas que contêm alguma ocorrência desta variável.

Para sabermos se uma cláusula é satisfeita, não é satisfeita ou se é unitária associa-se a cada cláusula um contador de literais $\alpha$ que se satisfazem e $\beta$ que não se satisfazem. Uma cláusula não se satisfaz se o $\beta$ é igual ao número de literais da cláusula. Uma cláusula se satisfaz se o $\alpha$ for maior ou igual a 1. Já a cláusula é unitária se o $\alpha = \beta - 1$, restando um literal a ser assinalado. Quando uma cláusula é declarada unitária, percorremos a lista de literais para

[4]Isto acontece, novamente, na descrição de idéia geral do algoritmo.

identificar o literal que não tenha nenhum valor assinalado. Quando um conflito é detectado e o *backtracking* realizado, devemos atualizar os contadores de todas as cláusulas em que aparece alguma variável que passa de assinalada para não assinalada. Este tipo de estrutura de dados é a base de construção das estruturas encontradas nos algoritmos do POSIT [69], Satz [59], Relsat [67] e GRASP [44].

As estruturas baseadas em contadores apresentam o inconveniente de que, a cada vez que um valor é atribuído à uma variável, devemos analisar todas as cláusulas em que aparece esta variável. É uma situação de alto custo quando se resolvem instâncias onde existem muitas ocorrências de uma mesma variável e naqueles algoritmos que armazenam cláusulas onde conflitos foram encontrados. Neste caso o número de ocorrências aumenta à medida que a busca avança. Uma situação melhor é aquela onde um valor é assinalado a uma variável e então identificando quais cláusulas se tornam unitárias e quais cláusulas passam a ser não satisfeitas. Esta situação é encontrada nos algoritmos com estruturas de dados *lazy*. Estas estruturas se caracterizam por manter, para cada variável, uma lista de um número reduzido de cláusulas onde esta variável aparece.

O algoritmo SATO [64] foi o primeiro a utilizar tais estruturas de dados. Posteriormente seguiu-se Chaff [27] com estruturas de dados utilizando duas sentinelas (*two watched literals*), uma melhoria às estruturas de SATO apresentando a vantagem do *backtracking* a um custo constante.

Para o esquema dos dois literais sentinelas (Chaff), temos dois apontadores a dois literais da cláusula. Não é imposta nenhuma ordem na escolha destes sentinelas e cada apontador pode mover-se em qualquer direção. Inicialmente as variáveis dos literais sentinelas não possuem nenhum valor assinalado e cada variável possui ainda uma lista de apontadores às ocorrências positivas da variável que são sentinelas positivas (*positive-watched*$(x)$) e uma lista de apontadores às ocorrências negativas da variável que são sentinelas negativas (*negative-watched*$(\neg x)$). Quando um valor 1 é assinalado a uma variável $x$, por exemplo, para cada ocorrência de $\neg x$ da lista *negative-watched*$(\neg x)$, o algoritmo busca um literal L na cláusula onde ocorre $\neg x$ e que não é avaliada em 0. Durante esta busca quatro situações podem ocorrer:

- se existir um literal L que não é o outro literal sentinela, eliminamos de *negative-watched*$(\neg x)$ o apontador a $\neg x$ e acrescentamos à lista de variáveis um apontador a L. Neste caso é dito que o literal sentinela foi movido.

- Se o único literal L que existe é o outro literal sentinela e a sua variável não possui nenhum assinalamento, então temos uma cláusula unitária cujo único literal é o outro literal sentinela.

- Se o único literal existente é o outro literal sentinela e é avaliado em 1, nada é feito.

- Se todos os literais da cláusula são avaliados em 0 e não existe nenhum outro literal L, a

cláusula é uma cláusula vazia e não é feito o *backtracking*[5]. Quando o algoritmo executa um *backtracking*, o custo de se desfazer o assinalamento de uma variável é constante. Isto é devido ao fato dos dois literais sentinelas serem os últimos cujas variáveis assinaladas os levaram a avaliarem como 0. Assim quando o *backtracking* é realizado não devemos modificar os apontadores destes literais porque são avaliados como 1 ou não possuem nenhum valor assinalado [58].

Na Figura 3.4 apresentamos as principais estruturas de dados *lazy*.

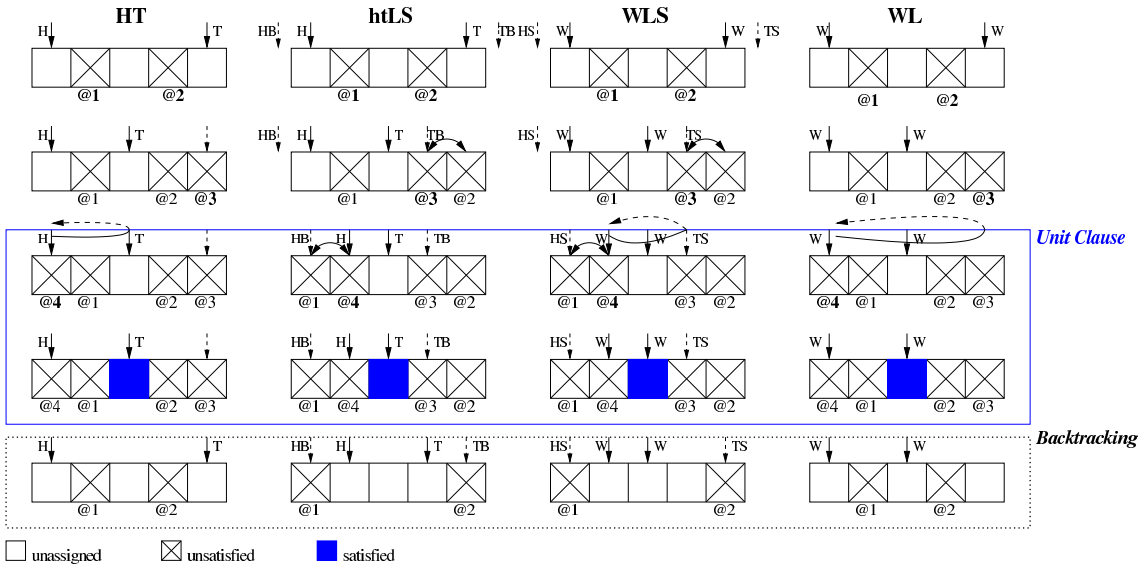


**Figura 3.4**: Estruturas *lazy* [49]

A primeira estrutura de dados *lazy* foi a chamada *Head/Tail*(HT) usada inicialmente por SATO. Esta estrutura de dados associa dois literais em cada cláusula como mostrado na primeira coluna da Figura 3.4. Inicialmente *head* aponta para o primeiro literal e *tail* aponta para o último literal. Cada vez que um literal assinalado é apontado pelo *head* ou *tail* (exceto quando o valor é 1), é buscado um novo literal ainda não assinalado. Identificado um literal não assinalado, este torna-se o novo *head* ou *tail*. Quando um literal satisfeito é identificado, a cláusula é declarada satisfeita. No caso de não se identificar um assinalamento do literal, a cláusula é declarada unitária, não satisfeita ou satisfeita dependendo do valor literal apontado pela outra referência (ou seja, se estivermos tratando do *head*, a outra referência será o *tail*). Quando o processo *backtrack* de procura é realizado, a referência que se tornou associada com o *head* ou *tail* pode ser descartada e uma nova referência se torna ativa (representada pela seta na coluna 1 da Figura 3.4).

[5] O *backtracking* é feito quando existe ainda algum literal não assinalado e o resultado encontrando anteriormente é 0.

O resolvedor Zchaff utiliza uma nova estrutura de dados chamada *Watched Literals* (WL) que resolve alguns problemas da lista HT (ocorre no processo de *backtrack* onde as referências dos *watched literals* não são modificados). Com a estrutura proposta em SATO, duas referências são associadas a cada cláusula. Entretanto em WL não há ordem relacionando as duas referências. A falta de ordem tem a vantagem de que nenhum literal de referência necessita ser atualizado quando o processo de *backtracking* começa. Já as cláusulas identificadas como unitárias ou não satisfeitas só são apresentadas depois de toda movimentação. A identificação de cláusulas satisfeitas é igual à da estrutura HT.

Na Figura 3.4 também são apresentadas duas outras estruturas de dados chamadas *Head-/Tail Lists with Literal Sifting* (htLS) e *Watched Literals with Literal Sifting* (WLS) que são outras formas de tratamento das estruturas de dados [41].

A heurística utilizada para selecionar as variáveis é outro aspecto importante nos algoritmos de satisfabilidade competitivos. O tamanho da árvore de pesquisa gerada, assim como o tempo de resolução, refletem os resultados desta heurística. Existe sempre um compromisso entre a redução do espaço de busca ocasionado pela escolha e o custo computacional para efetuar a escolha. Uma heurística utilizada é MOMS (*Most Often in Minimal Size clauses*). Esta heurística procura selecionar uma variável que mais ocorre entre as cláusulas. Intuitivamente estas variáveis irão permitir gerar mais cláusulas únicas durante a propagação unitária. Quanto mais cláusulas forem geradas maior a simplificação das fórmulas aumentando a possibilidade de se detectar antes um conflito ou chegarmos a uma cláusula vazia. Outra heurística utilizada é UP (*Unit Propagation*). Esta heurística explora melhor o potencial da propagação unitária que o MOMS. Para cada variável $x$ que ocorre em uma fórmula $F$, a *Unit Propagation* aplica a propagação unitária a $F \cup \{x\}$ e a $F \cup \{\neg x\}$ e assinala um peso à variável $x$ de acordo com os resultados obtidos ao aplicarmos a propagação unitária. Um efeito secundário é a capacidade de detectarmos literais falhos (*failed literals*): ao aplicarmos a propagação unitária a $F \cup x$ e detectarmos um conflito, fixamos o valor de $x$ a 0. O mesmo acontece se aplicarmos a propagação unitária a $F \cup \neg x$ e detectarmos um conflito então fixamos o valor de $x$ a 1.

Esta heurística permite gerar árvores de pesquisa menores que o MOMS. Entretanto tem o custo de cálculo maior. O algoritmo Satz utiliza uma heurística sofisticada através de uma combinação de MOMS e UP. O algoritmo Chaff [69] apresentou um novo enfoque no projeto das heurísticas de seleção de variáveis para algoritmos de satisfabilidade. A motivação dos autores era desenvolver uma heurística que fosse rápida, já que as heurísticas descritas são pouco eficientes nas fórmulas com grande número de cláusulas.

A heurística adotada é VSIDS (*Variable State Independent Decaying Sum*) que possui as seguintes características [66]:

- Cada variável possui um contador que está inicialmente em zero.
- Quando se inclui uma cláusula à base de dados de cláusulas, incrementamos o contador associado a cada um dos literais da cláusula. Deve-se levar em conta que se incluem

cláusulas tanto iniciais quanto as geradas durante a execução (aprendizagem de cláusulas com os conflitos).

- Escolhe-se a variável com maior contagem e então resolve-se os empates aleatoriamente.
- Periodicamente todos os contadores são divididos por uma constante (em geral 2). A idéia principal é escolher um literal que apareça muito em cláusulas incluídas mais recentemente em função dos conflitos encontrados [57].

Quando um conflito é detectado, o algoritmo DPLL executa um *backtracking* reiniciando a pesquisa por novas soluções retornando ao primeiro momento de decisão imediatamente anterior ao conflito encontrado. A este tipo de *backtracking* denominamos cronológico. Os algoritmos de satisfabilidade atuais (Relsat, SATO, GRASP, Chaff e BerkMin [24]) possuem um procedimento para a análise dos conflitos todas as vezes em que ocorrem. Este procedimento procura a razão do conflito e ao tentar resolvê-lo informa ao programa que não existe solução em uma determinada parte do espaço de busca e indica a partir de onde a busca deverá continuar. Normalmente este ponto é anterior ao encontrado pelo *backtracking* cronológico, reduzindo o espaço de busca que seria percorrido e que não levaria a nenhuma solução. A razão que levou ao conflito é identificada através de novas cláusulas geradas pelo próprio algoritmo e que são somadas às cláusulas iniciais. Estas cláusulas são redundantes mas evitam que se repita o caminho percorrido até a detecção de um conflito. Sem a inclusão destas cláusulas, novos conflitos tardariam a ser detectados em partes do espaço de pesquisa que não foram ainda explorados [24].

O tempo de CPU para resolução de problemas similares pode variar muito simplesmente alterando-se a ordem das variáveis. Este comportamento resultou na proposta de implementação de reinícios aleatórios. Intuitivamente podemos perceber que uma má seleção no espaço de busca pode levar à exploração de grandes partes do espaço de busca onde não existe solução. A proposta então é implementar o reinício da busca de tempos e dirigi-la para outra direção. Uma forma de alterarmos o rumo da pesquisa é introduzir uma aleatorização na heurística de escolha de variáveis. Desta forma quando se reinicia a busca, novas partes do espaço de busca serão percorridas. Nos algoritmos onde a aprendizagem de cláusulas é implementada, estas cláusulas criadas permanecem com as cláusulas iniciais quando a busca é reiniciada.

Apresentamos de forma geral as estruturas de dados principais das heurísticas para o problema SAT bem como o primeiro algoritmo utilizado (DPLL). Nos próximos itens deste Capítulo, explicaremos detalhadamente cada heurística com base na estrutura de dados discutida.

## 3.4 GRASP - *Generic Search Algorithm for Satisfiability Problem*

O objetivo dessa seção é explicar o funcionamento da heurística GRASP que visa unificar algumas técnicas de poda já propostas além de facilitar a identificação de novas cláusulas. A

análise de conflitos permite determinar onde o *backtrack* não-cronológico poderá ser executado em níveis inferiores fazendo grandes podas no espaço de procura. A gravação das causas de conflito, permitirá que a heurística reconheça ocorrências similares ao longo da procura.

### 3.4.1 Introdução

Começando de um conjunto vazio de variáveis assinaladas, o algoritmo de procura por *backtrack* organiza uma **árvore de decisão** onde cada nó elege um assinalamento para cada variável não-assinalada de acordo com a **decisão de assinalamento**. O **nível de decisão** é associado a cada decisão de assinalamento para denotar sua profundidade na árvore de decisão. A primeira decisão de assinalamento na raiz é chamado de nível de decisão 1. O processo de procura funciona da seguinte forma:

1. Estende-se o assinalamento atual fazendo uma decisão de assinalamento para uma variável não-assinalada. O **processo de decisão** é o mecanismo básico para explorar novas regiões no espaço de procura. A procura termina em sucesso se todas as cláusulas tornam-se satisfeitas.

2. Estende-se o assinalamento atual seguindo as consequências lógicas feitas anteriormente. Os assinalamentos adicionais derivados do **processo de dedução** estão relacionados ao **assinalamento de implicação**. Esse processo pode levar a identificação de mais cláusulas não-satisfeitas implicando que o atual assinalamento não é satisfeito. Tal ocorrência é denominado **conflito** e tal assinalamento é denominado **assinalamento conflituoso**.

3. Refaz-se o atual assinalamento devido ao conflito para que outro assinalamento de variáveis seja tentado. Esse processo de *backtracking* é o mecanismo básico para tratar novamente o espaço de procura por novos assinalamentos.

   O nível de decisão para cada variável dada $x$ é dada pela expressão $x = v@d$ que "significa $x$ se torna igual a $v$ no nível de decisão $d$".

### 3.4.2 Funções Importantes

Dada uma fórmula inicial $\varphi$, muitos sistemas de busca tentam aumentar o poder de dedução com implicações adicionais [38]. Isso está relacionado a uma função importante que chamaremos **aprendizagem** e pode ser executado tanto no pré-processamento como no processo de procura [37].

GRASP trabalha com aprendizagem no processo de procura que chamaremos de **aprendizagem dinâmica** baseada no diagnóstico de cláusulas conflitantes. Quando encontrado um conflito o processo de aprendizagem dinâmica tende a aprender com o erro que levou a tal conflito introduzindo novas implicações ao banco de dados. O **diagnóstico de conflito** produz três partes de informações que ajudam no *speed up* da procura:

1. Novas implicações que não existiam no banco de dados das cláusulas podem ser identificadas com a ocorrência do novo conflito. Essas cláusulas podem ser adicionadas ao banco de dados para evitar ocorrências futuras do mesmo conflito e representam CBE (*conflict based equivalence*).

2. Uma indicação se o conflito foi devido a mais recente decisão de assinalamento ou a anterior tal que:

   (a) Se o assinalamento é o mais recente, ou seja, no atual nível de decisão, o assinalamento oposto é imediatamente aplicado (se não tiver sido tentado). Chamaremos esse procedimento de DFC (*declaration failed caused*).

   (b) Se o conflito resulta pelo assinalamento anterior, ou seja, em outro nível de decisão, a procura realiza um *backtrack* ao nível correspondente. A capacidade de identificar o nível de *backtracking* do nível corrente de decisão é uma forma de *backtracking* não-cronológico que chamaremos de CDB (*conflict direct backtracking*) e é de suma importância para reduzir o espaço de procura por uma solução.

### 3.4.3 Estruturas do Processo de Procura

O mecanismo básico para derivar implicações do banco de dados de cláusulas é chamado de BCP (*Boolean Constraint Propagation*) [22] [63]. Seja dada uma fórmula qualquer $\varphi$ contendo a cláusula $\omega = (x + \neg y)$ e assumindo $y = 1$. Para que $\omega$ assuma o valor 1 e com isso $\varphi$ seja satisfeita, temos que ter $x = 1$. Logo temos que $y = 1$ implica que $x = 1$ devido a $\omega$. Chamaremos de **implicações lógicas** esse processo e corresponde a regra de cláusula unitária proposto por Davis-Putnam [19].

Seja o assinalamento da variável $x$ implicado pela cláusula $\omega = (l_1 + l_2 + ... + l_k)$. O assinalamento antecedente de $x$ é denotado por $A(x)$ que designa as variáveis assinaladas que são responsáveis diretas pela implicação do assinalamento de $x$ devido a $\omega$. A sequência de implicações gerada pelo BCP é representada pelo Grafo de Implicação I (veja Figura 3.5) definido como:

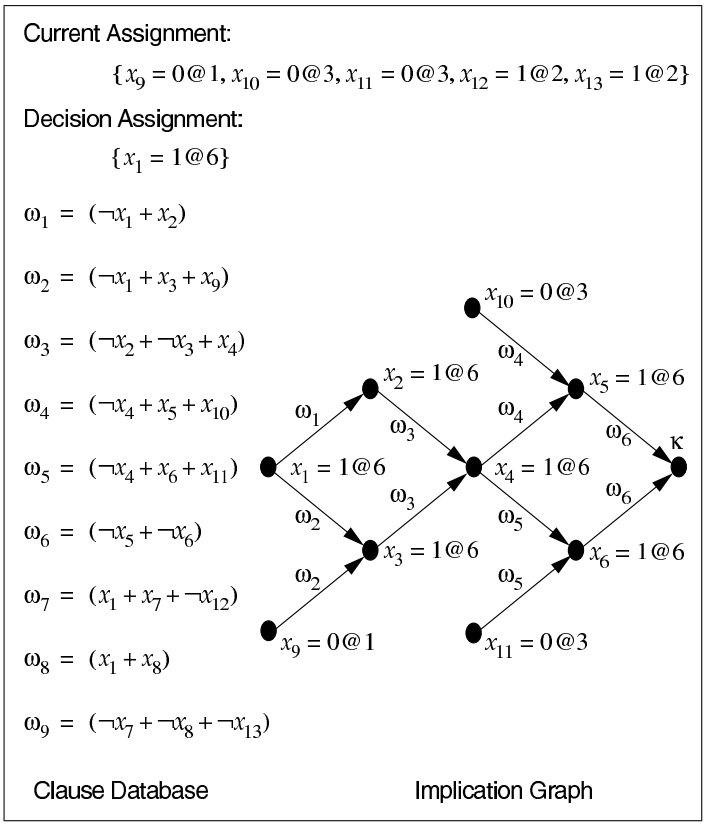


**Figura 3.5**: Grafo de Implicação I [45]

1. Cada vértice corresponde a um assinalamento de variável $x = \nu(x)$.

2. Os antecessores do vértice $x = \nu(x)$ são os assinalamentos antecedentes $A(x)$ correspondentes à cláusula $\omega$ que leva à implicação de $x$. Vértices que não têm antecessores correspondem às decisões de assinalamento.

3. Vértices de conflitos são adicionados para indicar a ocorrência de conflitos. Os antecessores do vértice conflituoso $\kappa$ correspondem ao assinalamento de variáveis que força a cláusula $\omega$ a se tornar não-satisfeita e é vista como o assinalamento antecedente $A(\kappa)$.

### 3.4.4 Heurística GRASP

A estrutura geral do GRASP pode ser visto na Figura 3.6. O procedimento recursivo de procura consiste em quatro operações maiores:

```
% Global variables    Clause database φ
%                     Variable assignment A
% Return value:       FAILURE or SUCCESS
% Auxiliary variable: Backtracking_level β
function GRASP ( ) {
  return (Search (0, β ) != SUCCESS) ?
          FAILURE: SUCCESS }
  % Input argument:    Current decision level d
  % Output argument:   Backtracking_level β
  % Return value:      CONFLICT or SUCCESS
end function
function Search (d, & β ) {
  if (Decide (d) = SUCCESS) then {
    return SUCCESS }
   while (true) {
    if (Deduce (d) != CONFLICT) then  {
       if  (Search (d + 1, β ) = SUCCESS) then {
          return SUCCESS }
       else if ( β  != d) then  {
          Erase ( )
          return CONFLICT }
       }
    }
  if  (Diagnose (d, β ) = CONFLICT) then {
        Erase ( )
        return CONFLICT }
    Erase ( )
   }
}
end function
function Diagnose (d, & β ) {
 ω c( κ ) = Conflict_Induced_Clause ( )
 Update_Clause_Database ( ω c( κ ) )
 β = Compute_Max_Level ( )
 if ( β  != d) then  {
      add new conflict vertex κ to I
      record A( κ )
      return CONFLICT }
 return SUCCESS
end function
```

**Figura 3.6**: Heurística GRASP [45]

1. Decide (), escolhe um assinalamento de decisão para cada estágio do processo de procura.

Procedimentos de decisão são baseados em heurísticas de conhecimentos. A heurística gulosa [6] é usada da seguinte forma: A cada nó na árvore de decisão é avaliado o número de cláusulas diretamente satisfeitas por cada assinalamento de cada variável. Escolhe-se a variável e o assinalamento que diretamente satisfaz o maior número de cláusulas.

2. Deduce (), implementa o BCP e mantém o resultado do grafo de implicação.

3. Diagnose (), identifica as causas de conflito e pode aumentar o banco de dados das cláusulas com implicações adicionais.

4. Erase (), apaga os assinalamentos necessários no nível de decisão em questão.

### 3.4.5 Análise de Conflitos

Quando um conflito surge do BCP, a estrutura da sequência de implicação convergindo no vértice $\kappa$ é analisada para determinar quais assinalamentos das variáveis são responsáveis pelo conflito [45]. A conjunção desses assinalamentos conflitantes é uma implicação que representa uma condição suficiente para que o conflito apareça. A sua negação produz uma implicação para a função Booleana que não existe no banco de dados da cláusula $\varphi$. Essa nova implicação, denominada **cláusula de conflito induzida**, fornece o mecanismo primário para implementação de falhas de declarações, *backtracking* não-cronológico e equivalência baseado em conflitos. Na Figura 3.7 observamos como é feito o *backtracking* não cronológico para a seguinte fórmula $\omega = \{\neg x_1 + x_9 + x_{10} + x_{11}\}$.

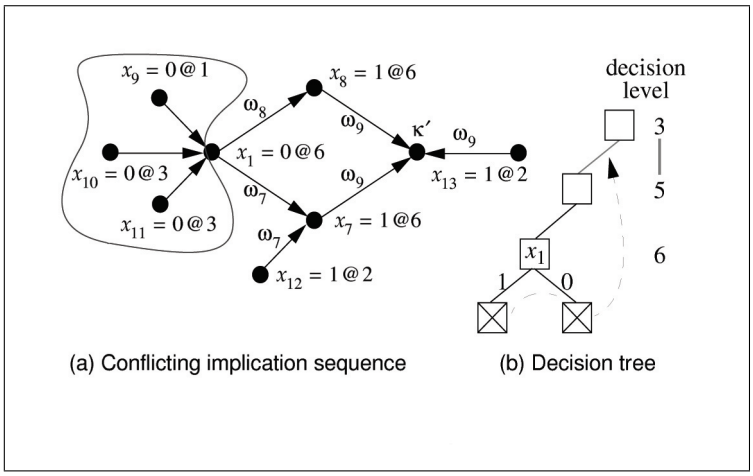


**Figura 3.7**: *Backtracking* não-cronológico [45]

[6] Os métodos de construção gulosa são técnicas muito utilizadas, que se baseiam no incremento da solução a cada passo, em que o elemento a incluir é determinado por uma função a que se dá o nome de função heurística. Em problemas de otimização, começa-se sem dar qualquer valor às variáveis, e que a cada passo atribui-se o valor a uma variável, por forma a que (para minimização) o aumento da função objetivo seja mínimo.

Se todos os literais correspondentes às variáveis que foram assinaladas num nível de decisão são menores que o corrente nível de decisão, concluimos que é necessário um *backtracking*. Essa situação só ocorre quando o conflito em questão é produzido como consequência direta do diagnóstico de um conflito anterior apresentado na Figura 3.7 (a). Já na mesma Figura parte (b) observamos que após a ocorrência do segundo conflito o nível de decisão do *backtrack* é calculado e se torna 3 (relativo às decisões apresentadas na Figura 3.5).

Na Figura 3.7 temos $x_{10} = 0$ pela decisão de nível 3. Na decisão 6 temos que $x_1 = 1$. Logo $x_2 = 1$ pela mesma decisão e $w_1$. Por $w_2$, $x_3$ assume o valor 1 pela mesma decisão. Por $w_3$, $x_4$ assume 1. Assim sucessivamente chegamos a um conflito, ou seja, não há nenhuma configuração que torna essa cláusula satisfeita.

### 3.4.6 Exemplo

Seja a seguinte fórmula normal conjuntiva (Figura 3.8):

$$(x_1 + x_2) * (\neg x_1 + x_2) * (\neg x_2 + x_3)$$

Se em determinado ponto da resolução, a heurística assumir que $x_2 = 0$, a cláusula $(\neg x_2 + x_3)$ será eliminada já que assumirá o valor 1. Temos então:

$$(x_1 + x_2) * (\neg x_1 + x_2)$$

Pelo grafo de implicações $x_1$ assume o valor 1 na cláusula $(x_1 + x_2)$ e $x_1$ assume o valor 0 na cláusula $(\neg x_1 + x_2)$ o que gera um conflito. Utilizando o *backtracking* não cronológico, retornamos ao nível da decisão anterior à que assume o valor $x_2 = 0$, gravando no banco de dados que a decisão $x_2 = 0$ gera conflito e apagando as decisões posteriores. Um outro caminho que a heurística poderia tomar, seria eleger $x_2 = 1$ eliminando as cláusulas $(x_1 + x_2) * (\neg x_1 + x_2)$. A fórmula se resumiria a $(\neg x_2 + x_3)$ e pelo grafo de implicação $x_3$ assumiria o valor 1, resultando numa fórmula satisfeita com a atribuição de $x_2 = 1$ e $x_3 = 1$.

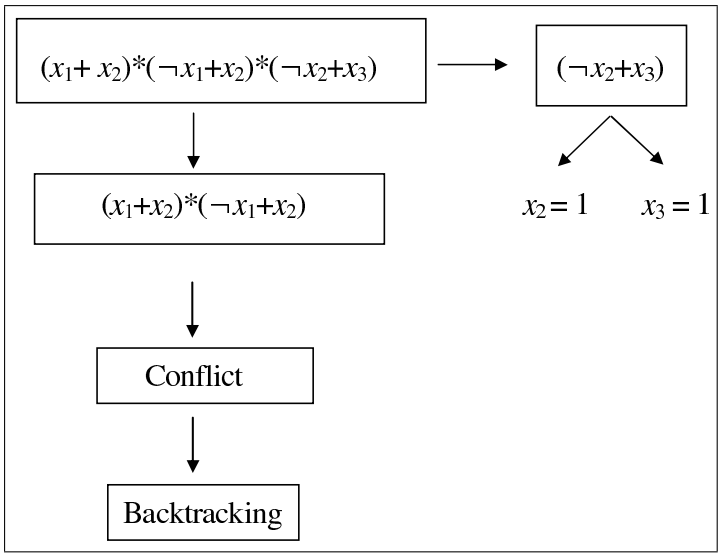


**Figura 3.8**: Exemplo da estrutura principal de GRASP

## 3.5 SATO - *SAtisfiability Testing Optimized*

### 3.5.1 Introdução

SATO é uma heurística desenvolvida no ano de 1996 por Hantao Zhang e Mark E. Stickel baseada no método proposto por Davis-Putnam em 1960 [67]. Uma das maiores motivações para o desenvolvimento dessa heurística é o de atacar problemas denominados *Latin square problems* [68].

SATO apresenta duas técnicas interessantes com o intuito de melhorar a performance do algoritmo Davis-Putnam. As regras de divisão (*splitting rules*) e a nova forma de se trabalhar conflitos, tornou SATO uma ferramenta muito importante e referenciada.

A escolha do literal a ser dividido é de importância fundamental em heurísticas que se baseiam em Davis-Putnam. É sabido que diferentes regras de divisão aumentam a performance do algoritmo Davis-Putnam. SATO fornece várias regras de divisão e por isso cada uma delas melhora o desempenho de determinadas classes de problemas SAT.

### 3.5.2 Definições Básicas

No estudo dos *Latin squares problems* uma regra de divisão tende a ser melhor que as demais: a escolha de um literal da menor cláusula positiva (uma cláusula positiva é uma cláusula onde todos os literais são positivos). Por outro lado foi provado [67] que uma regra de divisão eficiente é escolher uma variável $x$ tal que o valor $f_2(x) * f_2(\neg x)$ é máximo, onde $f(L)$ vale 1 mais o número de ocorrências do literal $L$ [22] [15].

SATO tenta combinar as duas regras. Seja $0 < a \leq 1$ e $n$ o menor número de cláusulas que

não são de *Horn* [7] no conjunto atual. Primeiramente coletamos todas os nomes das variáveis que aparecem no começo em $\lceil a * n \rceil$ da menor cláusula positiva. Depois escolhe-se $x$ nessa gama de variáveis com o máximo valor $f_2(x) * f_2(\neg x)$. Essa junção de regras funciona bem se $a$ é a porcentagem de cláusulas que não são de *Horn* na entrada multiplicado por cinco [66].

No BCP um literal é assinalado como Verdade tanto pelas regras de divisão como regra de propagação unitária [19]. Quando uma cláusula vazia é encontrada então se torna necessário um *backtracking*. Nesse ponto, SATO coleta todos os literais ativos que tiveram papel importante na cláusula vazia. Se o atual literal ativo não pertence a este conjunto, não é necessário tentar o outro valor dessa variável o que é chamado de **intelligent backjumping**.

Para evitar a coleta dos mesmos grupos de literais em estágios posteriores da procura, pode-se gravar a disjunção da negação desses literais coletados como uma nova cláusula no sistema. Entretanto essa nova técnica de se criar novas cláusulas nem sempre aumenta a performance. Em alguns casos a performance piora devido ao aumento de memória para o armazenamento dessas novas cláusulas.

### 3.5.3 Principal Função

A principal função introduzida em SATO é chamada de **trie-merge**. Comparada ao método de Davis-Putnam tal função apresenta várias vantagens:

1. Cláusulas duplicadas são automaticamente eliminadas;
2. A memória utilizada é reduzida devido a divisão de cláusulas prefixadas;
3. Cláusulas unitárias podem ser achadas rapidamente;
4. Cláusulas agrupadas pela extremidade são aquelas em que a primeira porção de literais também é uma cláusula. Por exemplo, $(x_1 + x_2 + x_3)$ é agrupado com $(x_1 + x_2)$. Uma das vantagens de SATO é que tais agrupamentos são eliminados na construção da função *trie-merge*.
5. *Resolução pela extremidade*: Sejam $(c_1 + L)$ e $(c_1 + \neg L)$ duas cláusulas. Uma resolução no último literal da primeira cláusula é o resultado $(c_1 + c_2)$ onde a segunda cláusula é suprimida. Ou seja, a segunda cláusula é substituída por $(c_1 + c_2)$.

Uma *trie* (também conhecida como *radix searching tree*) é uma árvore de busca na qual o fator de sub-divisão (*branching factor*), ou número máximo de filhos por nó, é igual ao número de símbolos do alfabeto.

Na Figura 3.9, apresentamos a função **trie-merge**.

[7] Uma cláusula de Horn é aquela que possui no máximo 1 literal positivo.

```
function trie-merge (trie t1, trie t2) return trie
    if (t1 = □ ) or (t2 = □ )then {
            return □ }
    else if (t1 = nil) then {
            return t2 }
    else if (t2 = nil) then {
            return t1 }
    t1 = <v1, p1, n1, r1>
    t2 = <v2, p2, n2, r2>
    if  (v1 = v2) then {
            n = trie-merge (n1, n2)
            r = trie-merge (r1, r2)
            return <v1, trie-merge (p1, p2),n,r> }
    else if (v1 < v2) then {
            return <v1, p1,n1,trie-merge (r1,t2)> }
    else {
            return <v2, p2,n2,trie-merge (r2,t1)> }
end function
```

**Figura 3.9**: Função *trie-merge* SATO [67]

Cada nó da árvore pode ser vazia ($nil$), pode apresentar uma marca final $\square$[8] ou uma tupla $< var, pos, neg, rest >$. Temos que $var$ é a variável indexada, $pos$ é o nodo positivo, $neg$ é o nodo negativo e $rest$ é o próximo nodo na lista linear que tem o mesmo pai [66]. No exemplo da Figura 3.11 temos o $pos$ representado pelo $+$, o $neg$ representado pelo $-$, o $rest$ representado pelo $DC$ e finalmente a $var$ representada pelo $x_1$.

A implementação mais bem sucedida do SATO é utilizando a função *trie-merge*. Na Tabela 3.1[9] podemos observar os resultados obtidos utilizando a implementação por meio de lista (SATO 1.2) e por meio da função *trie-merge* (SATO 1.3). Os resultados foram extraídos da referência [66]. Percebemos que os resultados utilizando *trie-merge* são melhores [56][65][10].

[8] Um nó vazio é aquele que não há ramificação, já aquele que apresenta a marca final indica que não há, partir deste ponto, mais ramificação (mas poderia ter acontecido anteriormente).

[9] As Tabelas apresentadas ao longo dessa dissertação são todas em relação ao tempo de execução em segundos.

[10] A classe de problemas utilizada para realização do teste é chamada de *Quasigroup* [47] e SATO foi desenvolvida especialmente para trabalhar com problemas desse tipo. Todas as tabelas apresentadas são medidas de tempo em segundos.

| Classe / Heurística | SATO 1.2(s) | SATO 1.3(s) |
|---|---|---|
| QG1.7 | 1.29 | 0.68 |
| QG2.7 | 1.04 | 0.62 |
| QG3.8 | 0.45 | 0.31 |
| QG4.8 | 0.39 | 0.28 |
| QG5.9 | 0.30 | 0.26 |
| QG6.9 | 0.16 | 0.16 |
| QG7.9 | 0.15 | 0.17 |

**Tabela 3.1**: Comparação de resultados de SATO

### 3.5.4 Exemplo

SATO incorporou os avanços da heurística GRASP o que significa que a forma de se tratar conflitos (usando Grafo de Implicação) e o uso *backtracking* não cronológico são utilizados nesta heurística. A grande diferença que determinou uma melhoria em SATO foi a utilização dos ponteiros *Head/Tail*. Seja a seguinte cláusula num nível de decisão $n$ qualquer (Figura 3.10):

$$(\neg x_1 + x_4 + \neg x_7 + x_{12} + x_{15})$$

Sabemos que há um ponteiro $H$ apontando para $x_1$ e um ponteiro $T$ apontando para $x_{15}$. Seja $x_1 = 1@1$ [11], logo o ponteiro $H$ mudará de posicão para $x_4$ já que a cláusula só será visitada se os ponteiros necessitarem de se mover (pois a cláusula não foi satisfeita). Seja $x_7 = 1@2$ e $x_{15} = 0@2$ então o ponteiro $T$ aponta para $x_{12}$ (em SATO *Head* apenas pode se mover em direção ao *Tail* e vice versa). Seja $x_4 = 0@3$ implicando que $x_{12}$ assuma o valor 1, tornando a cláusula satisfeita. Entretanto, suponha que no banco de dados das variáveis, este assinalamento de $x_{12}$ gera um conflito, então um *backtracking* é executado ao nível de decisão 1, apagando todas as decisões posteriores a este nível. A configuração atual é $H$ apontando para $x_4$ e $T$ apontando para $x_{15}$. Seja $x_{12} = 1@2$ e $x_7 = 0@2$, temos então que a cláusula assume o valor 1 e não é mais visitada. Já que encontramos o valor 1, os ponteiros de SATO não se movem mais.

[11] A variável $x_1$ assume o valor 1 pelo nível de decisão 1.

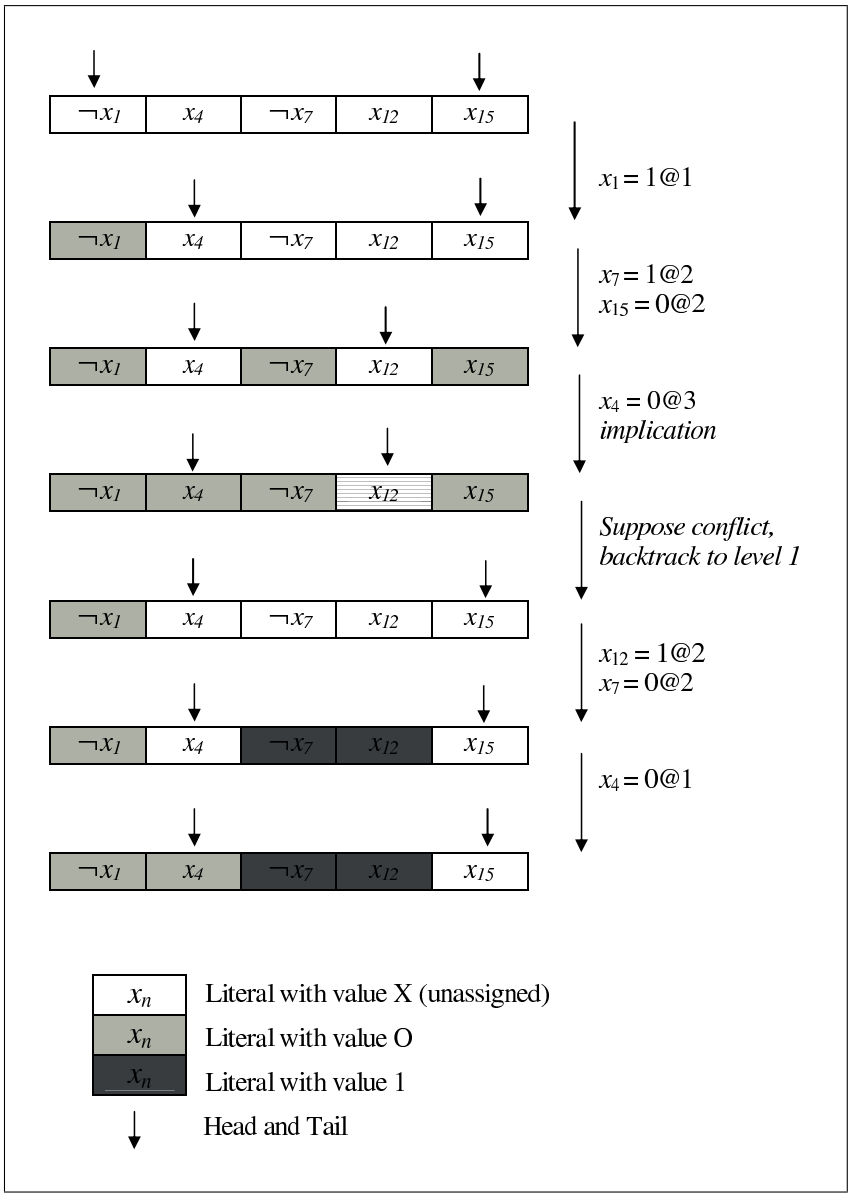


**Figura 3.10**: Exemplo da estrutura principal de SATO

Uma forma de representar a cláusula $(x_1 + x_2) * (\neg x_1 + x_3) * (\neg x_2 + x_3)$ pode ser vista na Figura 3.11[12].

[12]DC representa o *don't care*.

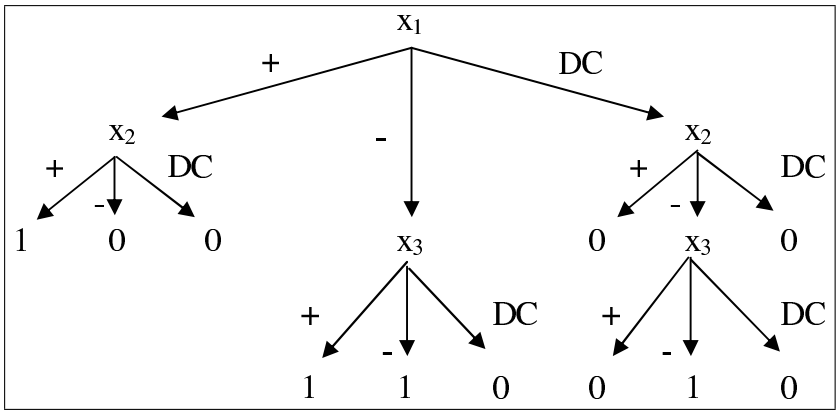


**Figura 3.11**: Exemplo *trie* SATO

## 3.6 Zchaff

Novamente baseado no mais popular algoritmo conhecido para resolução SAT (DPLL) [3], a presente heurística se baseia numa outra denominada Chaff [49]. Ela tende a melhorar a técnica utilizada por SATO utilizando literais sentinelas. Em contraste com as listas *Head* and *Tail* não há ordem entre os dois ponteiros. A falta de ordem tem a vantagem de que os literais não precisam ser atualizados quando são executados os *backtrackings*. Algumas extensões ao procedimento básico do Davis-Putnam foram bem sucedidas na heurística Zchaff, são elas:

1. Algumas regras de poda foram propostas em [32, 39, 45] e incorporadas com sucesso em Zchaff. A maioria dessas regras são variáveis de estado dependentes, isto é, são estatísticas de ocorrência de variáveis não-assinaladas usadas para marcar o banco de dados das cláusulas a fim de simplificar as varíaveis já assinaladas.

2. Em Zchaff adota-se o processo de aprendizagem de conflitos (*Conflict Learning*) aplicado primeiramente em Relsat [54] e posteriormente aprimorado em GRASP [44]. Esse processo evita que o algoritmo no espaço futuro de procura encontre os mesmos conflitos já mapeados.

3. O *backtracking* não-cronológico discutido em GRASP é utilizado também nessa heurística com o objetivo de aprimorar o *backtracking* tradicional proposto por Davis-Putnam [19].

4. Após abortar a execução devido ao excesso de algum parâmetro, reinicializações (*restarts*) são presentes nesse algoritmo. Esse processo evita que o algoritmo SAT se perca em partes não-relevantes do espaço de procura. O processo de aprendizagem afeta o conceito de reinicializações em cláusulas novas durante duas reinicilizações e então modifica o banco de dados das cláusulas.

### 3.6.1 Principais Funções

As principais funções relacionadas a tal heurística são:

1. **decide-next-branch:** Esta função implementa a regra de poda. Cada variável $x$ mantém o contador de literais (*literal count*) para contar o número de ocorrências de $x$ e de $\neg x$ em cláusulas pertencentes ao banco de dados. São mantidos também as contagens (*scores*) para cada $x$ e $\neg x$ individualmente. Todas as variáveis são mantidas em ordem decrescente com a função $maxcore(x) = max(score(x), score(\neg x))$ usada para ordenar a relação. Diante da aprendizagem de conflitos, o contador de literais pode ser aumentado e atualizações sempre são realizadas. Seja $x$ variável, $score_{old}(x), score_{old}(\neg x)$ são as atuais contagens e seja $increase(x), increase(\neg x)$ o incremento de $x$ e $\neg x$ desde a última atualização. Logo a nova contagem é realizada como:

$$score_{new}(x) = score_{old}(x)/2 + increase(x)$$

$$score_{new}(\neg x) = score_{old}(\neg x)/2 + increase(\neg x)$$

   Para uma nova ramificação de variáveis e um novo assinalamento de variáveis, a variável $x$ com o valor máximo $maxscore(x)$ é selecionado e seu assinalamento $x = 1$ se $score(x) \geq score(\neg x)$ e $x = 0$ se $score(x) < score(\neg x)$.

   Já que a contagem de literais somente é aumentada quando uma cláusula conflitante no banco de dados é adicionada, essa regra ocorre em cláusulas conflitantes aprendidas. Além disso, a contagem de variáveis que nunca ocorrem ou raramente ocorrem nas cláusulas conflitantes é dividida por dois.

2. **deduce:** Essa função incorpora a regra de propagação unitária (*unit propagation rule*) de uma maneira repetida e é chamada de BCP (*boolean constraint propagation*) já discutida anteriormente. A função principal do *deduce* é deduzir uma implicação de uma fila de implicações e checar todas as cláusulas não-satisfeitas se elas são agora satisfeitas ou se tornaram cláusulas unitárias a partir dessa nova implicação. Além disso, essa implicação é assinalada no nível de decisão nomeada no assinalamento anterior da decisão. Todos os assinalamentos de implicações para um nível de decisão são armazenados numa lista de anexada. Se um conflito ocorre ou seja, uma cláusula é identificada como não-resolvida, todas as cláusulas não-resolvidas são coletadas para permitir que a análise de conflitos baseada em cláusulas não-resolvidas. A função retorna **NO-CONFLICT** se nenhum conflito ocorre e retorna **CONFLICT** se ocorre conflito.

3. **analyze-conflicts:** Essa função ocorre dentro da função **deduce**. Um conflito ocorre quando um assinalamento contraditório na variável é deduzido. Por exemplo, seja $x$ uma variável que já tenha sido assinalada com o valor $x = 1$ e a partir da função **deduce**

encontramos o valor $x = 0$ o que caracteriza um conflito. Se $c = x$ temos que $c$ é uma cláusula conflitante. A partir do aprendizado através da cláusula conflitante o nível de *backtracking* é computado.

4. **back-track:** Depois da análise de conflitos é necessário realizar um *backtrack* no nível $b$ computado pela função **analyze-conflicts**. Logo todas as implicações e decisões de assinalamentos com nível de decisão $r \geq b$ são recompostas. Se dentro de uma cláusula conflitante $c$ somente um literal tiver seu nível de decisão, este literal é diretamente implicado. Esta implicação pode disparar novas implicações pela função **deduce** [27].

Na Figura 3.12 , apresentamos as funções principais do **Zchaff**.

```
function zChaff
    while (1) {
        if (decide_next_branch () ) then {
            while (deduce () = CONFLICT ) {
              blevel = analyze_conflicts () }
            if (blevel = 0) then {
                return UNSATISFIABLE }
            else {
                 back_track (level) }
        else {
            return SATISFIABLE }
    }
end function
```

**Figura 3.12**: O algoritmo Zchaff [27]

### 3.6.2 Literais Observados

Um dos principais avanços propostos, primeiramente por Chaff, e posteriormente incorporado em Zchaff é o que chamamos de **watched literals** [49].

Na prática, para a maioria dos problemas SAT, mais que 90% no tempo de execução de um resolvedor é responsável pelo processo BCP. Logo um eficiente BCP é de fundamental importância. Para implementar eficientemente o BCP, deseja-se achar uma forma rápida de visitar todas as cláusulas que se tornaram novas implicações por uma simples adição ao conjunto de assinalamentos.

A forma mais intuitiva é simplesmente olhar cada cláusula no banco de dados que está assinalada para 0. Deve-se ter um contador para cada cláusula que tenha o valor 0 e modificá-lo toda vez que um literal da cláusula for mudado para 0. Com esse objetivo, pode-se escolher dois literais não assinalados para 0 em cada cláusula para observar a qualquer momento. Então pode-se garantir que até que um desses dois literais sejam assinalados para 0, não se pode ter

mais que $N-2$ literais que na cláusula assinalados para 0 o que torna a cláusula não implicada. Ao se visitar um cláusula temos:

1. A cláusula não é implicada, então pelo menos dois literais não estão assinalados para 0, incluindo o outro literal observado corrente. Isto significa que pelo menos um literal não-observado não está assinalado para 0. Escolhe-se então este literal para substituir o que está assinalado para 0. Mantém-se a propriedade que dois literais observados não estão assinalados para 0.

2. A cláusula é implicada então deve-se notar que a variável implicada deve ser o outro literal observado, já que pela definição, a cláusula tem somente um literal nao assinalado para 0, e um dos dois literais observados é agora assinalado para 0.

Podemos observar na Figura 3.13 o funcionamento do BCP de Zchaff através de dois literais observados.

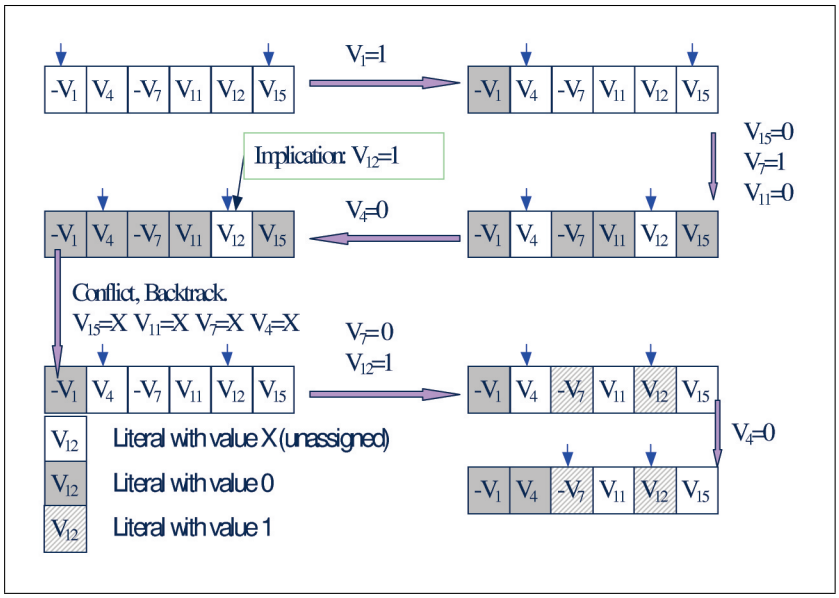


**Figura 3.13**: *Two Watched Literals* [42]

A Figura 3.13 mostra como os literais observados de Zchaff para uma cláusula simples muda de acordo com o assinalamento de suas variáveis. Observe que a escolha inicial dos literais observados é arbitrária. O BCP em SATO é bem parecido com o apresentado em Zchaff. A diferença básica entre Zchaff e SATO é o fato de que em Zchaff não é necessário o deslocamento numa direção fixa para os literais observados. Em SATO o *head* só pode movimentar-se em direção ao *tail* e vice versa. Na Figura 3.13 observamos que o os literais observados estão, a princípio, apontados para o início e fim da cláusula (o que não é necessário em Zchaff). Porém a medida que é executado o procedimento, não há mais direção fixa como aconteceria em SATO.

### 3.6.3 Resolvente

O principal avanço de Zchaff é a criação do que chamaremos resolvente com o intuito de prevenir erros no decorrer da heurística. Seja a CNF $(x_1 + x_2) * (\neg x_2 + x_3)$. Essa fórmula é equivalente a $(x_1 + x_2) * (\neg x_2 + x_3) * (x_1 + x_3)$ (pode ser verificado utilizando tabela verdade). O resolvente[13] da fórmula $(x_1 + x_2) * (\neg x_2 + x_3)$ será dado por $(x_1 + x_3)$ já que temos a implicação $(x_1 + x_2) * (\neg x_2 + x_3) \rightarrow (x_1 + x_3)$.

Por exemplo, seja $F = (a+b) * (\neg a + \neg b + \neg c) * (\neg b + c)$ e se tomarmos $a = 1$ não é executado o processo de dedução (já que na parte restante $(\neg a + \neg b + \neg c) * (\neg b + c)$ não se pode extrair nenhum resolvente.) Quando $b = 1$ ocorre um conflito entre $(\neg a + \neg b + \neg c)$ e $(\neg b + c)$. O resolvente será dado por $(\neg a + \neg b)$ o que evitará problemas futuros.

A verificação da veracidade dos resolventes se faz por meio do uso de tabelas verdades já que as fórmulas descritas em cada exemplo são equivalentes. Está "descoberta" de equivalências é que torna Zchaff uma heurística mais poderosa que as anteriores.

### 3.6.4 Exemplo

Zchaff incorporou os avanços da heurística GRASP e SATO o que significa que a forma de se tratar conflitos (usando Grafo de Implicação) e o uso *backtracking* não cronológico são utilizados nesta heurística além da idéia de literais observados. A grande diferença que determinou uma melhoria em Zchaff foi a utilização dos ponteiros que podem se mover livremente e, inicialmente, podem apontar para qualquer variável da cláusula.

Seja a seguinte cláusula num nível de decisão $n$ qualquer (Figura 3.14):

$$(\neg x_1 + x_4 + \neg x_7 + x_{12} + x_{15})$$

Sabemos que há dois ponteiros podendo apontar $x_7$ e para $x_{12}$. Seja $x_1 = 1@1$, logo os ponteiros não mudam de posicão. Seja $x_7 = 1@2$ e $x_{15} = 0@2$ então um ponteiro aponta para $x_4$ (em Zchaff os ponteiros se movem livremente) e o outro não muda de posição. Seja $x_4 = 0@3$ implicando que $x_{12}$ assuma o valor 1, tornando a cláusula satisfeita. Entretanto, suponha que no banco de dados das variáveis, $x_{12}$ gera um conflito, então um *backtracking* é executado ao nível de decisão 1 (nível inicial de acordo com o exemplo), apagando todas as decisões posteriores a este nível. A configuração atual é um ponteiro apontando para $x_4$ e outro apontando para $x_{12}$ já que quando é executado um *backtracking* em Zchaff não é necessário refazer o apontamento inicial dos ponteiros. Seja $x_{12} = 1@2$ e $x_7 = 0@2$, temos então que a cláusula assume o valor 1 e não é mais visitada. Em Zchaff, encontrando uma atribuição que faz com que a cláusula seja 1, os ponteiros se movem também (diferentemente de SATO).

[13] O resolvente é definido como uma tautologia que podemos extrair de determinada fórmula.

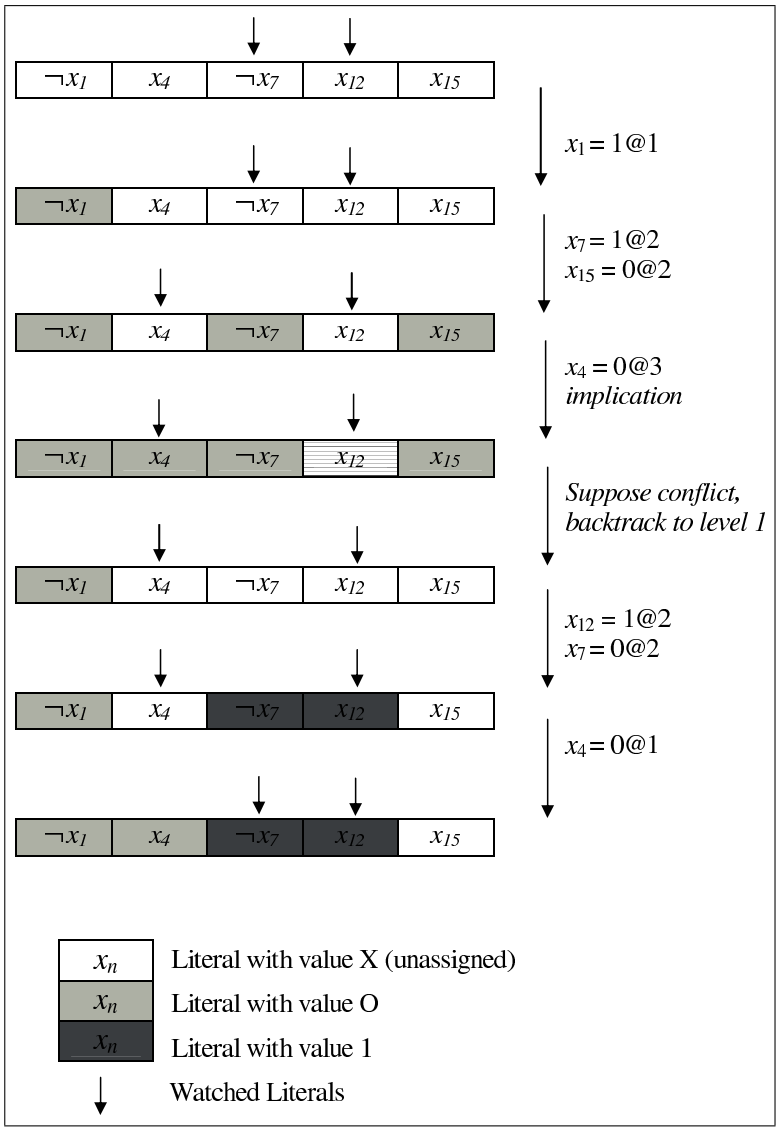


**Figura 3.14**: Exemplo da estrutura principal de Zchaff

## 3.7 Berkmin-Berkeley-Minsk

A presente heurística a princípio se mostrava mais robusta e resolvia mais problemas que até então melhor heurística conhecida Zchaff. No entanto artigos mais recentes não assumem que Berkmin é melhor que Zchaff embora apresente novidades importantes dentro dessa área de estudo.

Berkmin utiliza os procedimentos de *backtracking* não-cronológico e análise de conflitos introduzidos em GRASP, BCP sugerido em SATO e as decisões diante de conflito apresentadas em Zchaff. É um algoritmo completo baseado, também, em Davis-Putnam.

Novos aspectos na administração do banco dados de cláusulas além de diferentes tomadas

de decisão tornam Berkmin uma ferramenta muito eficiente para a solução de problemas SAT.

1. O conjunto de cláusulas conflitantes é organizado cronologicamente (a cláusula no topo é a que foi deduzida por último). Se o corrente nodo da árvore de procura é de cláusulas não-satisfeitas, a próxima ramificação de variáveis é escolhida entre as variáveis livres, dos quais os literais estão no topo das cláusulas conflitantes não-satisfeitas.

2. Apresenta-se uma heurística para escolher entre duas possibilidades de assinalamento para saber qual ramificação deve ser examinada primeiro.

3. O procedimento de se computar a atividade das variáveis é diferente de Zchaff. Zchaff usa um contador para contar o número de ocorrências de $x$ em cláusulas conflitantes o que pode levar a desprezar algumas variáveis que não aparecem nas cláusulas conflitantes e que contribuem para conflitos. Berkmin soluciona esse tipo de problema estimando um conjunto maior de variáveis envolvidas nas cláusulas conflitantes.

4. Um novo procedimento para administrar o banco de dados das cláusulas é apresentado em Berkmin.

### 3.7.1 Decisões de Berkmin

Para melhorar a performance dessa heurística alguns aspectos importantes foram pensados. O fato de que os resolvedores SAT acham uma solução rápida para a maioria das CNF´s que modelam problemas reais e o fato de que o conjunto das variáveis responsáveis pelo conflitos pode mudar dinamicamente, muda os paradigmas atuais. Isso implica que a escolha da ramificação para *problemas reais* deve ser dinâmica. Deve-se adaptar todas as mudanças no conjunto de variáveis devido a um novo assinalamento.

Berkmin implementa a seguinte função. Cada variável é assinalada a um contador ($ac(z)$) que armazena o número de cláusulas responsáveis por pelo menos um conflito que contenha o literal $z$. O valor de $ac(z)$ é atualizado durante o processo de BCP. Tão logo uma nova cláusula responsável por um novo conflito é encontrada, os contadores das variáveis, cujos os literais estão na cláusula, são incrementados de 1. Os valores de todos os contadores são periodicamente divididos por uma constante pequena maior que 1 (no caso do Zchaff essa constante é igual a 2 e no caso de Berkmin a constante é igual a 4) [24]. Dessa forma a influência de cláusulas *antigas* diminui e a preferência é dada às novas cláusulas deduzidas.

Em Zchaff somente literais em cláusulas conflitantes adicionadas à corrente CNF são consideradas quando computada a *atividade* da variável $z$ (então se o literal $z$ ocorre numa cláusula responsável pelo corrente conflito mas $z$ não está presente na cláusula conflitante, Zchaff não muda o valor da *atividade* de $z$).

Berkmin tem dois procedimentos de decisão. Primeiro escolhe a variável $z$ com o valor máximo de $ac(z)$ como a próxima ramificação, entretanto a principal decisão é baseada na ordenação cronológica de cláusulas conflitantes.

Embora apresente novidades teóricas, Berkmin não foi muito discutido em virtude de não apresentar resultados empíricos expressivos que mostrem a sua supremacia em relação à Zchaff.

### 3.7.2 Comparação de resultados

Berkmin apresentou melhores resultados em relação a Zchaff em algumas classes de problemas, porém de uma forma geral não podemos dizer que uma heurística seja melhor ou mais robusta que outra. Na Tabela 3.2 apresentamos classes de problemas onde a performance de Zchaff e Berkmin são bastante parecidas. Os resultados da Tabela 3.2 e da Tabela 3.3 foram retirados da referência [25]. Observe que em alguns problemas Berkmin apresenta melhores resultados e em outros Zchaff é melhor. É importante ressaltar que ambos resolveram todos os problemas, já que nenhum caso foi abortado.

| Classe / Heurística | Instâncias | Zchaff(s) | Berkmin(s) |
|---|---|---|---|
| Blockworld | 7 | 52.0 | 9.0 |
| Hole | 5 | 79.0 | 339.0 |
| Par16 | 10 | 33.6 | 13.6 |
| sss 1.0 | 48 | 41.6 | 13.4 |
| sss 1.0a | 8 | 17.2 | 17.7 |
| sss-sat 1.0 | 100 | 382.5 | 253.6 |
| fvp-unsat 1.0 | 4 | 589.0 | 1637.9 |
| vliw-sat | 100 | 2602.1 | 7300.5 |

**Tabela 3.2**: *Benchmarks* onde Berkmin se compara a Zchaff

Na Tabela 3.3 apresentamos classes de problemas onde a performance de Berkmin foi melhor que a de Zchaff. Observe que Berkmin resolveu todas as instâncias dos problemas[14] enquanto que Zchaff apresentou 84%[15] de sucesso.

| Classe / Heurística | Instâncias | Zchaff(s) | Berkmin(s) |
|---|---|---|---|
| Beijing | 16 | 468.0 (14) | 494.2 (16) |
| Miters | 5 | 1515.5 (3) | 3477.2 (5) |
| Hanoi | 3 | 1320.2 (2) | 1401.2 (3) |
| fvp-unsat 2.0 | 22 | 19678.2 (20) | 6869.3 (22) |

**Tabela 3.3**: *Benchmarks* onde Berkmin é melhor que Zchaff

De uma forma geral, os resultados atingidos por Zchaff e Berkmin são parecidos. Em algumas CNFs Zchaff executa mais rápido que Berkmin e em outras Berkmin[16] é melhor. Em ambos os casos a diferença das performances não é significante (a velocidade mais rápida não é superior

[14] O número entre parênteses nas tabelas representa as instâncias resolvidas.
[15] Observe que Zchaff não resolveu todas as instâncias do problema.
[16] Não foi apresentado um exemplo simples da heurística pelo fato de não estar disponível o código e o algoritmo.

à 20% ou mais que um minuto[17]). Entretanto Zchaff tem uma performance um pouco melhor globalmente enquanto que Berkmin apresenta resultados melhores em UNSAT CNFs.[18]

## 3.8 Resultados Empíricos Preliminares

Com o intuito de realizar os primeiros testes entre as heurísticas discutidas, apresentamos tabelas preliminares mostrando o tempo de resolução de cada heurística para determinada classe de problema [19]. Na próxima seção os resultados serão estendidos para diversas instâncias e outras classes de problemas.

Apresentamos, em forma de tabela, os resultados que através da nossa implementação conseguimos para as diferentes classes de problemas. Os experimentos foram realizados em um computador Pentium III, 700 MHZ com 256M.

### 3.8.1 *Graph Coloring Problem - GCP*

Dado um grafo $G = (V, E)$ onde $V = \{v_1, v_2, v_3, ...v_n\}$ é o conjunto dos vértices e $E$ é o conjunto de arestas que une os vértices achar uma coloração $C : V \rightarrow N$ tal que cada vértice tem uma cor diferente. Há duas variantes desse problema: na variante de otimização, o objetivo é achar uma colorãção com o mínimo número de cores e na variante de decisão é descobrir se com um número dado de cores é possível colorir o grafo [66].

A Tabela 3.4 mostra os resultados na classe GCP (*Graph Coloring Problem*) para os algoritmos estudados. SAT indica que o problema foi resolvido e satisfeito e UNSAT indica que o problema foi resolvido e não-satisfeito.

| SAT | Algoritmo | | |
|---|---|---|---|
| | Grasp | Sato | Zchaff |
| Tempo (seg) | 19.53 | 0.16 | 0.21 |
| #Cláusulas / #Literais | 2237 / 4674 | | |

**Tabela 3.4**: *Graph Coloring Problem*

Observamos claramente o melhor desempenho do algoritmo SATO que foi particularmente desenvolvido para se trabalhar com problemas da classe *Latin Square*. Entretanto, Zchaff não apresenta resultados ruins tanto no número de decisões quanto no tempo total de execução.

[17] Se observamos o tempo de resolução de cada instância do problema.

[18] Para uma análise bem detalhada e a apresentação de todos os resultados empíricos, veja referência [53].

[19] Alguns problemas foram explicados, porém devido a existência de muitos *benchmarks*, a maioria deles não será devidamente discutido.

### 3.8.2 *All Interval Series Problem - AIS*

Dado as doze notas musicais padrão $(c, c\#, d...)$, representadas pelos números $(0, 1..., 11)$ encontre uma série em que cada nota musical ocorre exatamente uma vez e em qual dos intervalos musicais entre notas vizinhas cubram o conjunto dos intervalos do segundo menor (1 semitom) ao sétimo principal (11 semitons). Isto é, para cada um dos intervalos, há um par de notas musicais vizinhas em que este intervalo aparece. O problema de encontrar tal série pode facilmente ser formulado como um exemplo de um problema aritmético mais geral em $Z_n$. Dado um $n$ em $N$, encontra um vetor $s = (s_1, ..., s_n)$, tais que (i) $s$ é uma permutação de $Z_n = \{0, 1..., n-1\}$; e (ii) o vetor $v$ do intervalo $= (|s_2 - s_1|, |s_3 - s_2|...|s_n - s_{n-1}|)$ é uma permutação de $Z_n - 0 = \{1, 2..., n-1\}$. Um vetor $v$ que satisfaz a estas regras é chamado de *All Interval Series Problem* de tamanho $n$.

A Tabela 3.5 mostra os resultados da classe AIS de tamanho $n = 8$ e a Tabela 3.6 mostra os resultados para o mesmo problema de tamanho $n = 10$.

| SAT | Algoritmo | | |
|---|---|---|---|
| | Grasp | Sato | Zchaff |
| Tempo (seg) | 0.04 | 0.01 | 0.02 |
| #Cláusulas / #Literais | 1520 / 3515 | | |

**Tabela 3.5**: *All-Interval Series - size 8*

| SAT | Algoritmo | | |
|---|---|---|---|
| | Grasp | Sato | Zchaff |
| Tempo (seg) | 6.16 | 0.02 | 1.1 |
| #Cláusulas / #Literais | 3151 / 7255 | | |

**Tabela 3.6**: *All-Interval Series - size 10*

Analisando os resultados obtidos pelas três heurísticas implementadas primeiramente para o problema *AIS* de tamanho 8, observamos que não há uma diferença considerável nos tempos de execução, já que todas resolveram o problema em tempos próximos. Entretando o número de decisões feitas por Zchaff é consideravelmente maior. Já o mesmo problema aplicado ao tamanho 10 apresentou um resultado muito bom para a heurística SATO, um resultado médio para Zchaff e um resultado bastante ruim para Grasp. Novamente o problema *AIS* pertence à classe *Latin Square* e por isso SATO apresenta resultados melhores. Observamos também que o número de decisões feitos por Zchaff é bem maior que as outras heurísticas. Tal fato ocorre já que Zchaff foi desenvolvido para se trabalhar com qualquer tipo de problema, apresentando resultados satisfatórios. Para problemas da classe *Latin Square*, apesar de se resolver em tempo razoável o número de decisões é maior que uma heurística desenvolvida principalmente para tal problema.

### 3.8.3 *Blocks World Planning Problem - BWP*

O BWP é um problema bastante conhecido em Inteligência Artificial. O cenário geral do BWP contém um número $n$ de blocos e uma mesa. Os blocos devem ser empilhados e há somente um operador que move o bloco superior de uma pilha para a mesa (observe que o ato de tirar um bloco de uma pilha necessariamente é também o ato de empilha-lo em outro lugar). Dado uma configuração inicial e final, o problema é achar a sequência de operações que aplicados à configuração inicial leva à configuração final. Os blocos só podem se mover quando não há nenhum bloco sobre ele.

A Tabela 3.7 mostra os resultados da classe BWP (*Blocks World Planning Problem*) para 9 blocos, e a Tabela 3.8 mostra os resultados para o mesmo problema com 11 blocos.

| SAT | Algoritmo | | |
|---|---|---|---|
| | Grasp | Sato | Zchaff |
| Tempo (seg) | 0.12 | 0.03 | 0.01 |
| #Cláusulas / #Literais | 4675 / 10809 | | |

**Tabela 3.7**: *Blocks World Planning Problem - size 9*

| SAT | Algoritmo | | |
|---|---|---|---|
| | Grasp | Sato | Zchaff |
| Tempo (seg) | 0.88 | 0.14 | 0.03 |
| #Cláusulas / #Literais | 13772 / 31767 | | |

**Tabela 3.8**: *Blocks World Planning Problem - size 11*

Analisando o resultado das heurísticas para o problema BWP observamos um resultado melhor de Zchaff. Comparativamente, Zchaff apresentou resultados bem melhores mostrando que essa heurística é realmente mais robusta que as outras. Observamos também que para esse problema o número de decisões feitas por Zchaff foi maior que as outras heurísticas porém relativamente menor que nas decisões das outras classes de problemas.

### 3.8.4 *Logistics Planning Problem - LPP*

Em LPP, pacotes devem ser transportados entre diferentes localidades de cidades diferentes. Os pacotes são transportados por caminhões dentro das cidades e na via aérea entre cidades. O problema envolve três operadores (*load, unload, move*) e dois predicados (*in, at*). O estado inicial e final especifica as localizações dos pacotes, caminhões e dos aviões. As ações podem ocorrer simultanêamente desde que não ocorram conflitos.

A Tabela 3.9 mostra os resultados obtidos para o problema LPP com 8 pacotes e 5 passos, e a Tabela 3.10 mostra os resultados obtidos para LPP com 5 pacotes e 13 passos.

| SAT | Algoritmo | | |
|---|---|---|---|
| | Grasp | Sato | Zchaff |
| Tempo (seg) | 7.65 | 0.26 | 0.03 |
| #Cláusulas / #Literais | 6718 / 17915 | | |

**Tabela 3.9**: *Logistics Planning Problem - size 5*

| SAT | Algoritmo | | |
|---|---|---|---|
| | Grasp | Sato | Zchaff |
| Tempo (seg) | 8.89 | 62.06 | 0.05 |
| #Cláusulas / #Literais | 7301 / 19024 | | |

**Tabela 3.10**: *Logistics Planning Problem - size 13*

Observamos empiricamente que o resultado de Zchaff para o problema *LPP* é bem superior que para as outras heurísticas. Observamos também que o número de decisões de SATO cresce drasticamente e por isso o tempo de execução para o problema *LPP* com 5 pacotes e 13 passos torna-se muito grande.

### 3.8.5 *A Pipelined DLX Processor*

Este *benchmark* é usado para validar a corretude de um *pipelined DLX Processor*. A Tabela 3.11 mostra os resultados para o problema *A Pipelined DLX Processor*.

| UNSAT | Algoritmo | | |
|---|---|---|---|
| | Grasp | Sato | Zchaff |
| Tempo (seg) | 9.93 | $> 12000$ | 0.10 |
| #Cláusulas / #Literais | 3725 / 10045 | | |

**Tabela 3.11**: *A Pipelined DLX Processor*

Zchaff apresentou um resultado muito bom para o problema do *Pipelined DLX Processor* enquanto que SATO foi abortado em virtude de ultrapassar o tempo estabelecido. Na verdade SATO não conseguiu resolver o problema e GRASP resolveu o problema com um tempo muito superior ao Zchaff.

### 3.8.6 *Pigeon Hole Problem - PH*

O problem PH pergunta se é possível colocar $n + 1$ pombos em $n$ buracos sem que dois pombos que estejam no mesmo buraco (obviamente, não é possível) [1].

A Tabela 3.12 mostra os resultados para o problema PH (*Pigeon Hole Problem*).

| UNSAT | Algoritmo | | |
|---|---|---|---|
| | Grasp | Sato | Zchaff |
| Tempo (seg) | 3.15 | 1.32 | 0.44 |
| #Cláusulas / #Literais | 297 / 648 | | |

**Tabela 3.12**: *Pigeon Hole Problem*

Novamente observamos o melhor desempenho de Zchaff comparativamente com as outras heurísticas no problema PH.

### 3.8.7 *Morphed Graph Coloring Problem*

Esta *benchmark* é apenas uma variação do problema GCP apresentado. A sua característica principal é a de transformar anéis de *lattices* [18] em *random* grafos.

A Tabela 3.13 mostra os resultados para o problema *"Morphed" Graph Colouring*.

| SAT | Algoritmo | | |
|---|---|---|---|
| | Grasp | Sato | Zchaff |
| Tempo (seg) | 1.69 | 0.02 | 0.01 |
| #Cláusulas / #Literais | 3100 / 6500 | | |

**Tabela 3.13**: *"Morphed" Graph Colouring*

Zchaff resolveu o problema *"Morphed" Graph Colouring* melhor que seus concorrentes embora SATO tenha apresentado um resultado também muito bom.

## 3.9 Resultados Comparativos

Com o objetivo de avaliar a melhor heurística disponível para diversas aplicações SAT, há uma competição organizada desde 2002 em conjunto com o Simpósio Internacional de Teoria e Aplicação de Provas de Satisfabilidade [59, 29, 23, 35]. São colocados à disposição dos participantes vários tipos de *benchmarks* para que sejam comparados os resultados obtidos. Há três tipos principais de classes de problemas:

1. Problemas Industriais: aqueles criados a partir de cenários reais encontrados no mercado.
2. Problemas Aleatórios: criado a partir de um critério mais amplo como número de variáveis e cláusulas envolvidas.
3. Problemas Preparados Manualmente: criados com o objetivo específico de explorar algumas nuances em determinadas fórmulas booleanas.

Com o objetivo de estender a aplicação dessas heurísticas, foram disponibilizados também problemas satisfatíveis e não satisfatíveis [20]. O objetivo dessa seção é o de analisar os principais resultados encontrados para diversas classes de problemas. Apresentaremos de forma didática (seção 3.9.1) quais heurísticas apresentam melhor resultado para vários tipos de problemas. As heurísticas foram implementadas e várias tabelas foram geradas com o intuito de analisar de uma forma mais geral os principais algoritmos[21].

### 3.9.1 Análise de Resultados

Abaixo apresentamos e analisamos os resultados comparativos das principais heurísticas.

| Classe/Heurística | DP | GRASP | SATO | SATO 1.3 | Zchaff |
|---|---|---|---|---|---|
| aim-100 | 240000.00 | 0.52 | 0.14 | 0.14 | 0.21 |
| aim-200 | 240000.00 | 8.29 | 0.45 | 0.35 | 0.70 |
| aim-50 | 172050.33 | 0.24 | 0.08 | 0.09 | 0.14 |
| ais | 40000.00 | 253.15 | 0.24 | 0.29 | 57.92 |
| morphed 8.0 | 1000000.00 | 23.03 | 1.99 | 1.90 | 3.83 |
| morphed 8.1 | 1000000.00 | 22.86 | 1.83 | 2.01 | 3.95 |
| Quasigroup | 220000.00 | 86544.53 | 1087.70 | 337.61 | 845.89 |
| Total | 54 m 10 s | 7 m 32 s | 18 m 12 s | 5 m 42 s | 15 m 12 s |

**Tabela 3.14**: Tabela dos Resultados *Latin Square*

A Tabela 3.14 apresenta os resultados para diferentes classes de problemas pertencentes à *Latin Square Problems*. Todas as heurísticas resolveram os problemas propostos. A heurística DP não resolveu no tempo proposto como o máximo nenhum dos problemas. Observe a implementação de duas heurísticas SATO. A versão de SATO 1.3 utiliza a função *trie-merge* e por isso apresenta melhores resultados. Os resultados encontrados para GRASP não são bons comparados com as outras heurísticas. GRASP não utiliza as estruturas de dados *head-tail* apresentadas em SATO e nem *two watched-literals* apresentada em Zchaff. Para essa classe de problemas, SATO apresentou resultados melhores que Zchaff concluindo que para *Latin Square Problems* a estrutura *head-tail* utilizando *trie-merge* apresenta melhores resultados.

[20] É de fundamental importância essa separação de problemas, já que em algumas aplicações não é importante, por exemplo, problemas não satisfeitos como *Bounded Model Checking [52]*.

[21] O código dos algoritmos estão disponíveis na *internet*. Os algoritmos do item *Resultados Preliminares* foram implementados pelo autor bem como àqueles resultados empíricos que não apresentarem referência. Os resultados também podem ser gerados de forma remota como os apresentados no item *Análise de Resultados*.

| Classe / Heu | DP | GRASP | SATO |
|---|---|---|---|
| avg-check (6) | 40612.39 (2) | 20137.95 (4) | 20060.34 (4) |
| avg-check-simp (6) | 40273.98 (2) | 20083.23 (4) | 20009.32 (4) |
| avg-check-tcsimp (6) | 40400.56 (2) | 20109.70 (4) | 19303.67 (5) |
| des-encryption (32) | 211185.22 (11) | 104352.59 (22) | 80402.84 (24) |
| eq-checking (34) | 160586.73 (18) | 36.11 (34) | 10007.25 (33) |
| longmult (16) | 140011.19 (2) | 82309.19 (9) | 22270.98 (15) |
| Total (100) | 51 m 10 s (37) | 37 m 8 s (77) | 47 m 34 s (85) |

**Tabela 3.15**: Tabela 1 dos Resultados de problemas industriais

| Classe / Heu | SATO 1.3 | Zchaff |
|---|---|---|
| avg-check (6) | 20013.28 (4) | 124.39 (6) |
| avg-check-simp (6) | 20016.03 (4) | 77.46 (6) |
| avg-check-tcsimp (6) | 20014.44 (4) | 114.10 (6) |
| des-encryption (32) | 104695.16 (23) | 22726.43 (30) |
| eq-checking (34) | 10053.80 (33) | 2.45 (34) |
| longmult (16) | 7590.25 (16) | 4502.49 (16) |
| Total (100) | 39 m 42 s (84) | 39 m 7 s (98) |

**Tabela 3.16**: Tabela 2 dos Resultados de problemas industriais

As Tabelas 3.15 e 3.16[22] mostram problemas industriais importantes. Observamos que o tempo gasto para resolver estes problemas é bem superior aos utilizados para resolver os problemas da Tabela 3.14. Notamos que algumas heurísticas com Davis-Putnam (DP) e GRASP, não conseguiram resolver um porcentagem considerável de instâncias (no caso de DP apenas 37% foram resolvidos enquanto que em GRASP 77% das intâncias foram resolvidas). Em tempo total de execução a heurística SATO 1.3 e Zchaff apresentaram resultados parecidos (ambas em torno de 39 minutos) porém o sucesso da resolução de Zchaff foi de 98% enquanto que SATO 1.3 teve 84 % de sucesso. Logo, empiracamente, observamos que para problemas industriais Zchaff apresenta resultados consideravelmentes melhores que as outras heurísticas. Em alguns *benchmarks*, SATO 1.3 saiu-se melhor que Zchaff, porém na média, Zchaff alcançou melhores resultados.

[22] Os números apresentados entre parênteses são as instâncias dos problemas e quantos problemas cada heurística resolveu.

| Classe / Heu | DP | GRASP | SATO |
|---|---|---|---|
| Beijing (16) | 151022.98 (1) | 34978.47 (13) | 44078.88 (12) |
| hole(5) | 40639.76 (1) | 11520.25 (4) | 180.50 (5) |
| ssa (8) | 60021.63 (2) | 2.68 (8) | 1.59 (8) |
| ucsc-bf(223) | 2060116.50 (17) | 84.18 (223) | 63.83 (223) |
| ucsc-ssa(102) | 940034.69 (8) | 25.88 (102) | 22.62 (102) |
| Total (354) | 17 m 15 s (29) | 56 m 51 s (350) | 19 m 7 s (350) |

**Tabela 3.17**: Tabela 1 dos Resultados de problemas especiais

| Classe / Heu | SATO 1.3 | Zchaff |
|---|---|---|
| Beijing (16) | 23814.70 (14) | 20268.12 (16) |
| hole(5) | 81.64 (5) | 42.81 (5) |
| ssa (8) | 8508.05 (8) | 0.89 (8) |
| ucsc-bf(223) | 84620.58 (216) | 22.28 (223) |
| ucsc-ssa(102) | 97525.28 (94) | 6.83 (102) |
| Total (354) | 35 m 50 s (337) | 39 m 0 s (354) |

**Tabela 3.18**: Tabela 2 dos Resultados de problemas especiais

As Tabelas 3.17 e 3.18 apresentam resultados interessantes que serão analisados. Esses problemas foram especialmente elaborados para se testar heurísticas (problemas inventados ou matematicamente elaborados). Observamos que Zchaff apresenta soluções para **todas** instâncias dos problemas no menor tempo. Notamos que a heurística SATO 1.3, que apresentou os melhores resultados para *Latin Square Problems* e bons resultados para problemas industriais, apresentou resultados muito ruins em tempo de execução e também em porcentagem de instâncias resolvidas (o fato de resolver 95% instâncias não é um bom resultado já que GRASP tem 98% de sucesso juntamente com SATO). Alguns tempos de execução de GRASP e SATO são parecidos o que mostra que SATO não é uma boa heurística para essa classe de problema.

### 3.9.2 Conclusões

A partir de inúmeros resultados podemos apresentar a Tabela 3.19. Na procura de uma solução para um problema que não sabemos necessariamente a qual classe pertence, sugerimos que as heurísticas SATO, Zchaff ou Berkmin sejam utilizadas. Entretanto, podemos fixar um tempo de execução pequeno para determinada heurística não gaste mais do que seja necessário.

| Classe / Heurística | DP | GRASP | SATO | SATO 1.3 | Zchaff | Berkmin |
|---|---|---|---|---|---|---|
| Latin Square | Não | Não | Sim (1) | Sim | Não (2) | Não (3) |
| Industriais | Não | Não | Não | Não | Sim | Sim (4) |
| Matemáticos | Não | Não | Não | Não | Sim | Sim |
| Qualquer Problema | Não | Não | Sim (5) | Sim (6) | Sim (7) | Sim (8) |

**Tabela 3.19**: Comparação entre Heurísticas

**Observações - Legenda:**

1. Apesar de recomendar seu uso, a utilização de SATO 1.3 apresenta melhores resultados.
2. Pelo fato de SATO apresentar melhores resultados comprovados, o uso de Zchaff não é recomendável.
3. Pelo fato de SATO apresentar melhores resultados comprovados, o uso de Berkmin não é recomendável. Embora não tenha sido feita uma comparação direta, há tabela comparativa entre Zchaff e Berkmin mostrando que são muito parecidos seus resultados.
4. Já que esta heurística é comparável a Zchaff, recomendamos também seu uso.
5. Apresenta resultados razoáveis.
6. Apresenta resultados razoáveis.
7. Sempre apresenta bons resultados na média.
8. Resultados razoáveis [25].

## 3.10 Comparações - Resumo

Abaixo apresentamos de forma resumida as principais heurísticas e suas particularidades.

### 3.10.1 GRASP

- Grafo de Implicações;
- Processo *Learning* de novas cláusulas;
- Vantagens: *Non-chronological backtracking*!
- Desvantagens: Não apresenta resultados bons devido ao seu BCP.

### 3.10.2 SATO

- SATO: *head and tail pointers*. Cada cláusula tem 2 apontadores *Head* aponta para o primeiro literal da cláusula; *Tail* aponta para o último literal da cláusula.
- Vantagens: Quando o ponteiro assume o valor 0 o literal apontado pelo *head* ou *tail* troca de posição; Quando a variável assume valor 1 as cláusulas que contém essa variável não são visitadas, o que diminui o tempo de execução.
- Desavantagens: Não é considerado robusto já que só resolve bem os problemas da classe *Latin Square*.

### 3.10.3 Zchaff

- *Chaff*: *Two watched literal pointers*. Não há ordem para os ponteiros como no SATO;
- Cada ponteiro pode se mover em qualquer direção.
- Mais robusto que o SATO.

## 3.11 Considerações finais

Neste capítulo apresentamos as principais heurísticas desenvolvidas com o objetivo de atacar o problema SAT. Mostrou-se que o conhecimento prévio de cada heurística serviu para aprimorar e melhorar as performances das mais recentes. Apresentamos também diversos resultados empíricos com o objetivo de demonstrar os bons resultados alcançados no tratamento de SAT.

Verificou-se empiricamente que a heurística Zchaff apresenta resultados bem melhores que GRASP e SATO. Observamos que para a classe de problemas *Latin Square* SATO apresentou os melhores resultados embora Zchaff tenha resolvido tais problemas com tempo de execução satisfatório. Em todas outras classes de problemas, Zchaff apresentou resultados muito superiores que as outras heurísticas resolvendo todos com tempo de execução muito superior aos seus concorrentes, mostrando-se mais robusto que os outros.

Podemos dizer também que as Heurísticas Zchaff e Berkmin apresentam os melhores resultados. Em algumas classes, Berkmin se saiu melhor que Zchaff, porém em outros Zchaff saiu-se melhor. Apesar destas nuances, ambas conseguiram resolver a maioria dos problemas e por isso podemos dizer que são heurísticas robustas.

# Capítulo 4

# Lógica Multivalorada

*Eu vi uma Roda altíssima, que não estava nem em frente aos meus olhos, nem atrás, nem ao meu lado, mas em todos os lugares ao mesmo tempo. Essa Roda era feita de água, mas também de fogo, e era (mesmo que eu pudesse ver sua borda) infinita.*
*Jorge Luis Borges*

## 4.1 Introdução

Muitos problemas de decisão em tarefas CAD (*Computer-aided design*) podem ser formulados como problemas de satisfação em um espaço multidimensional. Neste Capítulo apresentamos uma breve introdução à Lógica Multivalorada utilizando dois tipos de codificação: *One-hot-encoding* (OHE) e *Double-rail* [17, 40, 18].

## 4.2 Definições

1. $x_i \in X$ denota uma variável multivalorada no domínio $P_i = \{0, 1, ...|P_i| - 1\}$.

2. Uma variável é dita assinalada no conjunto $v_i$ se $x_i$ pode assumir qualquer valor de $v_i \subseteq P_i$

3. Um literal multivalorado $x_i^{s_i}$ é uma função booleana definida por:

$$x_i^{s_i} \equiv (x_i = \gamma_1) + ... + (x_i = \gamma_k)$$

   onde $\gamma_j \in s_i \subseteq P_i, j = 1, 2, ..., k$. Temos que $s_i$ denota o conjunto de valores dos literais. Isso quer dizer que se temos uma variável assinalada como $x_2^{\{1,3\}}$, $x_2$ será verdade se e somente se assumir o valor 1 ou o valor 3.

4. A valoração de uma variável multivalorada de acordo com a variável assinalada $x_i^{s_j}$ no

conjunto de valores $v_i$ é uma função multivalorada definida como:

$$V(x_i^{s_j}) \mid_{v_i} = \begin{cases} 1 & \text{se} \quad v_i \cap s_j = v_i \\ 0 & \text{se} \quad v_i \cap s_j = 0 \\ X1 & \text{se} \quad v_i \cap s_j = s_j \\ X2 & \text{nos outros casos} \end{cases}$$

Temos que se $V = 1$ significa veracidade, $V = 0$ significa falsidade e $V = X1$ ou $V = X2$ significa que o valor do literal é desconhecido.

5. Uma cláusula multivalorada é uma disjunção de um ou mais literais multivalorados. Uma cláusula multivalorada pode ser expressa como:

$$\omega_k = \sum x_i^{s_j}$$

Por exemplo

$$\omega_2 = (x_3^{\{1,3\}} + x_2^{\{1,4,5\}} + x_5^{\{0\}})$$

Como sabemos, uma cláusula assumirá o valor 1 se e somente se um de seus literais assumir o valor 1, de outra forma assumirá o valor 0.

6. Uma resolução MV (multivalorada) combina duas cláusulas multivaloradas $\omega_1 = \sum x_i^{s_j} + x^s$ e $\omega_2 = \sum x'^{s'_j}_i + x^{s'}$ numa nova cláusula $\omega_{res} = \sum x_i^{s_j} + \sum x'^{s'_j}_i + x^{s' \cap s}$ chamada cláusula **resolvente**. E $x$ refere-se à variável resolvente[1].

7. Se um literal da cláusula assumir o valor $X1$ ou o valor $X2$ e os literais remanescentes forem falsos então o literal não assinalado será denominado literal unitário, e sua cláusula correspondente será unitária. Um conflito ocorre quando todos os literais numa cláusula assumem o valor falso e então temos um claúsula conflitante.

8. Uma fórmula está na forma normal conjuntiva, ou forma normal conjuntiva multivalorada (MV SAT), quando a representamos como uma conjunção de um conjunto de cláusulas multivaloradas. Sabemos então uma fórmula é satisfeita se e somente se todas as cláusulas assumem o valor 1. Logo o problema SAT é dito **satisfeito** se a fórmula assume o valor 1 e o problema é dito **não satisfeito** se encontramos um valor diferente de 1 no nosso conjunto de soluções.

Observamos que a codificação de um problema multivalorado é mais compacta. Por exemplo, seja a cláusula $\omega = (x_1^{\{1,3\}} + x_2^{\{1,4,5\}})$. Para representá-la em uma cláusula binária faríamos a seguinte cláusula: $\omega^b = (x_{11} + x_{13} + x_{21} + x_{24} + x_{25})$.

[1] O apóstrofo denota o complemento.

## 4.3 Lógica Binária Estendida

O uso da lógica binária estendida está relacionado com os problemas nas quais o sistema binário precisa ser estendido. Em particular, naquelas situações onde necessitamos avaliar uma expressão com a presença de entradas desconhecidas, se algumas entradas não afetam a saída desejada podemos mantê-las como desconhecidas. Quando estamos trabalhando com entradas desconhecidas, consideramos o sistema como extensão do sistema binário já que o terceiro valor (X) pode assumir o valor tanto de 1 como 0. Trabalhando com lógica ternária, teríamos o sistema pertencente ao conjunto $\{0, 1, 2\}$ como definido no item anterior.

Dentre as possíveis aplicações podemos citar problemas cotidianos como acesso ao crédito. Para um sistema de análise de crédito, diferentes questionamentos são realizados com o objetivo de verificar se o candidato está apto ou não ao crédito. Podemos obter respostas que atendam às necessidades (1), que não atendam às necessidades (0) e respostas em branco (X).


**Figura 4.1**: Representação para $(1, X, 0)$

Podemos observar através da Figura 4.1, a representação de $(1, X, 0)$ utilizando *lattices*. Para construirmos os operadores lógicos, utilizamos os conceitos de *Greatest Lower Bound* (GLB) e *Least Upper Bound* (LUB). Para o operador AND, utilizamos o GLB e para o operador OR utilizamos o LUB. Isso significa que para fazermos o AND entre 1 e 0 por exemplo, utilizamos o conceito GLB que diz que a maior cota inferior ganha, o que faz com que seja o 0 (como na lógica binária sabemos). Já se fazemos o OR entre 1 e 0, utilizamos o conceito LUB que diz que a menor cota superior ganha, o que faz com que seja o 1.

As Tabelas 4.1, 4.2 e 4.3 mostram a tabela verdade para uma porta *AND*, *OR* e *NOT* utilizando lógica binária estendida.

| AND | 0 | 1 | X |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | X |
| X | 0 | X | X |

**Tabela 4.1**: AND para Lógica Binária Estendida

| OR | 0 | 1 | X |
|---|---|---|---|
| 0 | 0 | 1 | X |
| 1 | 1 | 1 | 1 |
| X | X | 1 | X |

**Tabela 4.2**: OR para Lógica Binária Estendida

| NOT | 0 | 1 |
|---|---|---|
| | 1 | 0 |
| | X | X |

**Tabela 4.3**: NOT para Lógica Binária Estendida

Utilizando os conceitos de *double-rail-logic* podemos identificar três estados lógicos para os sinais e dois bits para identificar cada sinal. A Tabela 4.4 apresenta a forma de trabalhar com lógica binária estendida utilizando dois bits.

| A | $A_1$ | $A_0$ |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |
| X | 1 | 1 |

**Tabela 4.4**: Representação de Lógica Binária Estendida utilizando dois bits

Para atender aos postulados da Álgebra Boolena (Lei Cumulativa, Lei Distributiva, Identidade e Complemento) consideramos as seguintes regras:

1. $AND(a, b) \rightarrow (a_1 b_1, a_0 + b_0)$
2. $OR(a, b) \rightarrow (a_1 + b_1, a_0 b_0)$
3. $NOT(a) \rightarrow (a_0, a_1)$

Tais regras são definidas para a realização de operações lógicas utilizando a representação *double-rail-logic* para lógica binária estendida. Observe que as variáveis $a$ e $b$ representam os valores $1, 0, X$ sendo descritos através de dois bits. O AND entre 1 e 0, pode ser escrito como o

AND$(10, 01)$ e fica sendo o $a_1$ AND $b_1$ que dá 0 e $a_0$ OR $b_0$ que dá 1. Encontramos então o valor 0 representado por 01.

A Figura 4.2 apresenta a representação *double-rail* para as portas $AND, OR$ e $NOT$ utilizando as regras acima.

| **And** | 01 | 10 | 11 |
|---|---|---|---|
| 01 | 01 | 01 | 01 |
| 10 | 01 | 10 | 11 |
| 11 | 01 | 11 | 11 |

| **Or** | 01 | 10 | 11 |
|---|---|---|---|
| 01 | 01 | 10 | 11 |
| 10 | 10 | 10 | 10 |
| 11 | 11 | 10 | 11 |

| **Not** | 01 | 10 |
|---|---|---|
| | 10 | 01 |
| | 11 | 11 |

**Figura 4.2**: Representação *Double-rail*

A fim de aplicar os conceitos introduzidos, representamos um fórmula normal conjuntiva em circuitos combinatórios. Como cada porta lógica em um circuito está relacionada com uma fórmula lógica que deve ser satisfeita independentemente, podemos extrair uma fórmula característica da saída de qualquer circuito lógico (ou subcircuito) iniciando pela saída e caminhando em direção à entrada tomando a conjunção de todas as fórmulas encontradas para os nodos visitados. Como a fórmula para cada porta lógica deve ser independentemente satisfeita, a conjunção das fórmulas também deve ser satisfeita. Utilizando a representação *double rail*, podemos construir CNF´s para representar estados quaisquer como a Figura 4.3.

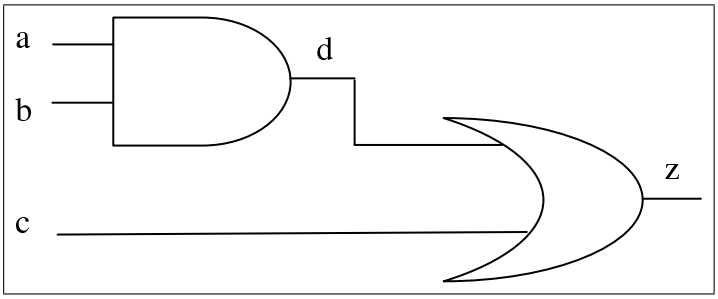


**Figura 4.3**: Representação de CNF em um circuito

As equações para cada porta lógica e a equação para a saída do circuito da Figura 4.3 podem ser escritas da seguinte forma:

$$AND : (\neg d + a) * (\neg d + b) * (d + \neg a + \neg b)$$

$$OR : (z + \neg c) * (z + \neg d) * (\neg z + d + c)$$

$$OUT : (\neg d + a) * (\neg d + b) * (d + \neg a + \neg b) * (z + \neg c) * (z + \neg d) * (\neg z + d + c)$$

Como resultado podemos obter as fórmulas CNF que serão submetidas aos resolvedores SAT.

Ao analisarmos os estados lógicos em um circuito podemos obter:

1. Circuitos Combinatórios com estados iniciais bem conhecidos: dois estados lógicos $(0, 1)$ que permitem computar de forma fácil o comportamento lógico.

2. Circuitos sequenciais com estado inicial desconhecido. Apresenta três estados lógicos $(0, 1, X)$ onde $X$ representa o estado inicial de *flip-flops* e células de memória *Ram*.

Podemos então analisar um circuito lógico através de um solucionador SAT e detectar variáveis que tornam a saída 1, 0 ou X. Quando um circuito contendo milhares de portas lógicas for analisado basta acrescentarmos as variáveis desconhecidas ao conjunto das fórmulas de satisfabilidade do sistema para ver se o resolvedor SAT binário estendido consegue ou não chegar a uma resposta. A Figura 4.4 mostra esquematicamente o funcionamento de um resolvedor SAT. Sabemos o valor de algumas variáveis de entrada $(A, B)$ , entretanto não são conhecidos os valores de outras variáveis $(C, D)$. O objetivo final é determinarmos se o circuito $(K)$ irá se comportar de acordo com uma especificação fornecendo uma saída desejada $(f)$.

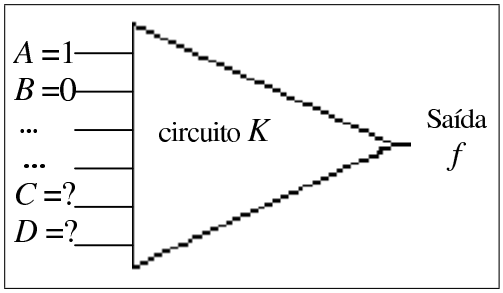


**Figura 4.4**: Resolvedor SAT para Lógica Binária Estendida

Outro tipo de codificação muito interessante que será utilizada no próximo Capítulo é a chamada *One-hot-enconding* (OHE). Para representar cada valor apenas um bit é ativado. Por exemplo, os valores $(1, X, 0)$ podem ser representados utilizando OHE como: $(001, 010, 100)$. Nas Tabelas 4.5, 4.7 e 4.6 apresentamos os operadores e suas representações OHE.

| AND | 001 | 010 | 100 |
|---|---|---|---|
| 001 | 001 | 010 | 100 |
| 010 | 010 | 010 | 100 |
| 100 | 100 | 100 | 100 |

**Tabela 4.5**: AND para codificação OHE

| OR | 001 | 010 | 100 |
|---|---|---|---|
| 001 | 001 | 001 | 001 |
| 010 | 001 | 010 | 010 |
| 100 | 001 | 010 | 100 |

**Tabela 4.6**: OR para codificação OHE

| NOT | |
|---|---|
| 001 | 100 |
| 010 | 010 |
| 100 | 001 |

**Tabela 4.7**: NOT para codificação OHE

## 4.4 Lógica Ternária

A Lógica Ternária nada mais é que uma Lógica Multivalorada com 3 valores. A diferença básica entre a ternária e a binária estendida é o fato que $X$ pode assumir o valor 0 ou 1. Um resolvedor SAT desenvolvido para o caso binário estendido, analisa o valor de uma fórmula normal conjuntiva, que pode ser 1, 0 ou $X$. Quando não conseguimos resolver o problema com um número qualquer de literais que assumem o valor $X$, a saída será $X$, ou seja, a fórmula normal conjuntiva não pode assumir nem o valor 0 nem o valor 1.

Na Lógica Ternária temos, por exemplo, 3 valores (1, 2, 3) que podem ser representados como *lattices*. A Figura 4.5 apresenta esta estrutura.


**Figura 4.5**: Lógica Ternária

Os operadores AND, OR e NOT podem ser representados como nas Tabelas 4.8, 4.9 e 4.10.

| AND | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 1 | 2 | 3 |
| 2 | 2 | 2 | 3 |
| 3 | 3 | 3 | 3 |

**Tabela 4.8**: AND para Lógica Ternária

| OR | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 2 | 1 | 2 | 2 |
| 3 | 1 | 2 | 3 |

**Tabela 4.9**: OR para Lógica Ternária

| NOT | |
|---|---|
| 1 | 3 |
| 2 | 2 |
| 3 | 1 |

**Tabela 4.10**: NOT para Lógica Ternária

Um exemplo de uma fórmula normal conjuntiva para lógica ternária (1,2,3) pode ser escrita como:

$$\omega = (x_3^{\{1\}} + x_2^{\{1,2\}}) * (x_4^{\{1,3\}} + x_3^{\{1\}} + x_1^{\{1\}}) * (x_{1,2}^{\{2\}} + x_1^{\{2,3\}} + x_4^{\{3\}})$$

Se $x_1 = 1$ e $x_2 = 1$ temos que a fórmula $\omega$ é satisfeita.

## 4.5 Lógica Quaternária

A lógica tetravalente mais conhecida é a de Belnap cujos valores não estão organizados como $1 - 2 - 3 - 4$, linearmente ordenados, mas segundo o *lattice* mais simples $1 - 2$, $1 - 3$, $2 - 4$, $3 - 4$. Nesta lógica a conjunção e a disjunçao mais usuais são definidos utilizando os conceitos de GLB e LUB, e a negaçao é tal que $\neg 1 = 4$, $\neg 4 = 1$, $\neg 2 = 2$, $\neg 3 = 3$. Os valores designados como "verdadeiros" são 1 e 2, e na realidade há duas ordens usuais sobre os valores, uma que reflete o aumento da verdade, e outra o aumento do conhecimento. A Figura 4.6 representa a lógica tetravalente de Belnap [2].

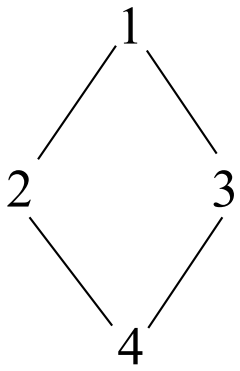


**Figura 4.6**: Lógica Quaternária de Belnap

## 4.6 Considerações finais

Nesse capítulo introduzimos o conceito e notação para lógica multivalorada. Apresentamos dois tipos de codificação bastante interessantes no tratamento de lógica binária estendida e multivalorada. Uma breve explanação a respeito de lógica ternária e quaternária foi também apresentada. Mostramos, através de figuras, como trabalhar com essa codificação através de portas lógicas.

# Capítulo 5

# Algoritmos Revisitados

*"Always be drunk*
*...that's it! ...the great imperative!*
*in order not to feel time's horrid*
*burden bruise your shoulders,*
*grinding you into the earth,*
*get drunk and stay that way*
*...on what? ...on wine, poetry,*
*virtue, whatever ...but get drunk.*
*Baudelaire*

## 5.1 Introdução

Com base nos algoritmos apresentados no Capítulo 3 e com as noções de lógica multivalorada apresentadas no Capítulo 4, apresentamos dois algoritmos que resolvem instâncias multivaloradas do problema SAT.

Primeiramente apresentaremos um algoritmo binário estendido baseado na heurística Davis-Putnam. Apresentamos uma tabela de resultados para avaliar seu funcionamento. Depois apresentaremos uma nova heurística conhecida por CAMA [40] que utiliza a codificação OHE multivalorada. CAMA foi desenvolvida utilizando os aprimoramentos das heurísticas de GRASP, SATO e Zchaff além de utilizar novas análises com o objetivo de melhorar seu desempenho.

## 5.2 Davis-Putnam Binário Estendido

O algoritmo utilizado na resolução do problema SAT binário estendido é o mesmo Davis-Putnam porém acrescentando um novo teste referente ao valor da variável. Antes da tentativa de uma nova variável, todos os valores possíveis para aquela variável são avaliados. Para resolver o problema $(1, X, 0)$ utilizamos o Davis-Putnam binário estendido e para resolver o problema $(0, 1, 2)$ ou qualquer outro multivalorado, utilizamos o CAMA. Abaixo apresentamos o algoritmo Davis-Putnam modificado para lógica $(1, X, 0)$ [5].

Apresentamos na Figura 5.1 o Davis-Putnam binário estendido utilizado para resolver o problema $(1, X, 0)$[1].

```
function Satisfiable (clause set S) return Boolean
    % unit propagation
    repeat
        for each (unit clause L in S) do {
                choose a literal assignment in L
                delete (L+Q) from S
                delete ¬L from (¬L+Q)∈S }
        if (S is empty) then {
                return true }
        else {
                if (a clause becomes null in S) then {
                for each (unit clause L in S) do {
                    (choose a different literal assignment in L)
                    (delete (L+Q) from S)
                    (delete ¬L from (¬L+Q)∈S) }
                if (S is empty) then {
                    return false }
        }
    until (no further changes result)
    % splitting
    (choose a literal L occurring in S)
    (choose a literal assignment in L)
    if (Satisfiable (S ⋃ {L})) then {
            return true }
    else if (Satisfiable (S ⋃ {¬L})) then {
            return true }
    else {
            return false }
end function
```

**Figura 5.1**: Davis-Putnam Binário Estendido

### 5.2.1 Resultados

O algoritmo da Figura 5.1 foi implementado em dois circuitos de teste, uma ALU simples e um circuito multiplicador, utilizados como geradores das cláusulas CNF. Os seguintes resultados foram obtidos (Tabelas 5.1 e 5.2). O critério adotado para escolha de variáveis foi selecionar a variável que aparecia o maior número de vezes nas cláusulas restantes a serem avaliadas.

[1] A diferença deste algoritmo para o apresentado no Capítulo 3 é o fato de testarmos outro valor para o literal e ver se o problema é resolvido atribuindo 0 ou 1 onde antes só poderia ser atribuído um destes valores. No caso de se resolver o problema com 0 e 1, então o literal é $X$.

| Multiplicador bit x bit | OHE(s) | *Double-rail*(s) |
|---|---|---|
| 3 | 0.045 | 0.134 |
| 4 | 0.216 | 0.618 |
| 5 | 0.778 | 1.69 |
| 6 | 2.31 | 5.58 |
| 7 | 6.259 | 10.12 |
| 8 | 15.958 | 24.24 |

**Tabela 5.1**: Resultados de um Multiplicador para Davis-Putnam Binário Estendido

| ALU | OHE(s) | *Double-rail*(s) |
|---|---|---|
| 2 | 0.06 | 0.21 |
| 8 | 0.995 | 3.00 |
| 16 | 7.664 | 24.00 |
| 32 | 64.345 | 208.00 |
| 64 | 529.34 | 2312.4 |

**Tabela 5.2**: Resultados de uma ALU para Davis-Putnam Binário Estendido

Nas Tabelas 5.1 e 5.2 observamos os resultados das implementações em *software* do algoritmo de Davis-Putnam em versão *double-rail* e em multivalor OHE. O melhor desempenho da codificação OHE se dá pela maior velocidade de acesso a memória [2].

O tempos necessários à solução do problema na codificação OHE apresentam resultados inferiores à solução do problema em codificação *double-rail*. O uso de *backtracking* não-cronológico, aprendizado de cláusulas e literais observados podem ser usados com o intuito de melhorar a perfomance do Davis-Putnam binário estendido.

## 5.3 Um resolvedor SAT para Lógica Multivalorada

Apresentamos os passos principais para a concretização de um resolvedor SAT multivalorado eficiente denominado CAMA [40]. A sua heurística é baseada no resolvedor binário Zchaff descrito no Capítulo 3 para SAT.

No caso de variáveis multivaloradas, a parte chave da heurística de decisão é determinar o número ótimo de valores assinalados. Existem dois casos extremos: (1) A decisão escolhe exatamente um valor do conjunto de variáveis ou (2) a decisão exclui um valor do conjunto. O primeiro caso denotado por *large decision scheme* leva imediatamente a um assinalamento completo de variáveis para cada decisão. O segundo caso denotado por *small decision scheme* usa uma aproximação que refina a decisão corrente incompleta removendo um valor do conjunto de variáveis.

[2]Para passar de 1 para 0, apenas dois bits são mudados.

A vantagem do primeiro caso é o pequeno número da profundidade de decisões com uma rápida redução no potencial do espaço de soluções. Por outro lado, já que a procura é restrita a um pequeno conjunto de soluções, as cláusulas de conflito formadas são fracas e contém pouca informação a respeito da procura futura, que é utilizado pela heurística CAMA.

O segundo caso aumenta o potencial de procura de uma solução tornando mais complexo tal procedimento e tornando mais forte as cláusulas de conflito formadas. No entanto, a vantagem é que excluído um valor por decisão, será necessário visitar somente essas cláusulas.

## 5.4 Análise de Conflitos de CAMA

O procedimento de análise de conflito multivalorado utilizado pelo resolvedor CAMA apresenta um aprimoramento em relação aos outros procedimentos apresentados no Capítulo 3.

Seja os seguintes literais não-resolvidos :

$$\omega_1 = (x_3^{\{1\}} + x_2^{\{0,1\}})$$

$$\omega_2 = (x_4^{\{1,3\}} + x_3^{\{0\}} + x_1^{\{1\}})$$

$$\omega_3 = (x_2^{\{2\}} + x_1^{\{2,3\}} + x_4^{\{3\}})$$

Assuma o conjunto solução em que as variáveis podem assumir valores $\{0, 1, 2, 3\}$. Seja $x_4 = 0$ no nível 1 e $x_3 = \{3\}$ no nível 2. Na Figura 5.2 é mostrado o respectivo grafo de implicação. Assume-se que as implicações são feitas da esquerda para a direita.

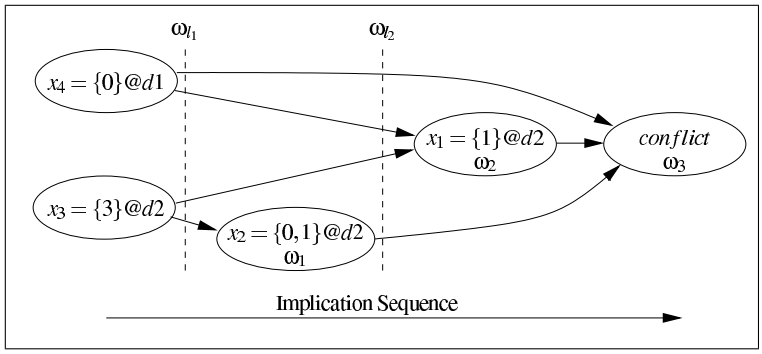


**Figura 5.2**: Grafo de Implicação II [40]

O processo tradicional de aprendizado coleta todos os literais não resolvidos e combina seu complemento para formar a seguinte cláusula conflitante:

$$\omega'_{l1} = (x_4^{\{1,2,3\}} + x_3^{\{0,1,2\}})$$

Esta cláusula é suficiente para implicar $x_3^{\{0,1,2\}}$ pelo assinalamento de $x_4 = \{0\}$. Entretanto a

seqüência abaixo utilizada no procedimento de análise de conflitos proposto pelo CAMA chega a uma cláusula mais forte[3].

$$\omega_3 = (x_2^{\{2\}} + x_1^{\{2,3\}} + x_4^{\{3\}})$$
$$\omega_2 = (x_4^{\{1,3\}} + x_3^{\{0\}} + x_1^{\{1\}})$$
$$\overline{\omega_{l2} = (x_2^{\{2\}} + x_4^{\{1,3\}} + x_3^{\{0\}})}$$

A notação adotada é a que os antecedentes são apresentados sobre a linha e sua dedução após a linha. Observe que a resolução apresentada acima elimina o literal $x_1$. A variável $x_1$ é implicada por $\omega_2$ com o resultado falso do literal $x_1^{\{2,3\}}$ de $\omega_3$ causando conflito. Pela construção, $x_1$ é descartado pois o conjunto de valores de $x_1$ em $\omega_3$ e $\omega_2$ não são sobrepostos (ou seja, não existe intersecção). O próximo passo é o seguinte:

$$\omega_{l2} = (x_2^{\{2\}} + x_4^{\{1,3\}} + x_3^{\{0\}})$$
$$\omega_1 = (x_3^{\{1\}} + x_2^{\{0,1\}})$$
$$\overline{\omega_{l1} = (x_4^{\{1,3\}} + x_3^{\{0,1\}})}$$

Agora o literal $x_2$ foi eliminado. Observe então que o $\omega_{l1}$ é mais forte que o $\omega'_{l1}$. Pela adição de $\omega_{l1}$, o assinalamento $x_4 = \{0\}$ implica $x_3 = \{0, 1\}$ que adicionado ao assinalamento conflituoso $x_3 = \{3\}$ também elimina o caso $x_3 = \{2\}$ apesar nunca ter sido assinalado anteriormente.

Na Figura 5.3 apresentamos o pseudo-código para a implementação de dois literais observados.

[3]O objetivo é explicar através do uso de cláusulos o procedimento de CAMA para análise de conflitos e suas melhorias comparadas à heurística Zchaff na qual ela foi inicialmente inspirada.

```
W_i: Set of watch literals associated to variable x_i
T_j: Set of non-false literals in clause j
function two_MV_lit_watch (x_i)
   for each (l_ij ∈ W_i) do {
     while (evaluate (l_ij) = false)  {
          if (T_j = 0) then {
             identify_conflict (j)
             return }
          else if (|T_j| = 1) then {
              identify_implication (T_j)
              break }
           else {
              W_i = W_i \ {l_ij }
              l_kj = get_next (T_j)
              W_k = W_k ⋃ {l_kj }
              break }
      }
   }
end function
```

**Figura 5.3**: Pseudo-Código MV-literais observados [40]

Na Figura 5.4 apresentamos um algoritmo para avaliar um literal multivalorado.

```
function evaluate (l)  return Boolean
   x = get_variable (l)
   imp_val_set = x.val_set ⋂ l.val_set
  if (imp_val_set = x.val_set) then {
     return true }
  if (imp_val_set = 0) then {
     return false }
  if (imp_val_set = l.val_set) then {
     return X1 }
  return X2
end function
```

**Figura 5.4**: Algoritmo para avaliar um literal multivalorado [40]

Observe que o procedimento **evaluate** 5.4 é chamado em cada passo do processo BCP. Já que é uma das funções mais frenquentemente chamadas, ela deve ser implementada eficientemente. É necessário observar se há redundância (por exemplo a implicação $x_2 = \{1, 3, 4\}$ é redundante se existir um atribuição anterior $x_2 = \{1, 3\}$) ou não a fim de otimizar o procedimento. O valor do literal unitário permite que o procedimento de dedução reconheça essa redundância e fuja dessa implicação. Uma valoração $X1$ pode ser usada para identificar a atribuição da implicação que

não é afetada pela atribuição anterior das variáveis. Essa situação ocorre quando a atribuição final da implicação é idêntica à implicação imediata.

Como mostrado anteriormente, o processo de resolução pode eliminar valores do conjunto de valores de cláusulas implicadas que não foram responsáveis pelo conflito atual. A técnica MVS (minimum value set) ajuda a identificar esses valores tal que o valor do conjunto de valores de cada literal multivalorado é mínimo. A claúsula resultante reflete a condição suficiente para o conflito e pode ser construida pela visita à cláusulas conflitantes. Na Figura 5.5 descrevemos o processo de análise de conflitos utilizado no algoritmo CAMA.

```
A_d : Assignment stack at decision level d,
Ψ : Set of marked variables,
Θ : Set of variables in the learned clause,
V(ω) : Set o variables in clause ω
Σ_d : Set of variables assigned at decision level d in clause ω.
function conflict_analysis (ω_c) return Boolean
   d = get_max_dec_level (ω_c)
   if (d = 0 ) then {
        return FAILURE }
    b = d
    Ψ = Σ_d (ω_c)
    Θ = V(ω_c) \ Σ_d (ω_c)
   for each (assignment α ∈ A_d ) do{
        x_α = get_variable (α)
        if (x_α ∈ Ψ ) then {  % skip unrelated assignments
          Ψ = Ψ \ { x_α } }
          if (Ψ = 0 ) then  {  % identify UIP
               Θ = Θ ⋃ { x_α }
               % new clause from Θ and the x.resolve_set's
               ω_l = create_learned_clause ( Θ )
               b = get_max_dec_level ( ω_l)
               imply ( ω_l)  % add to dataset and do BCP
               break }
         ω_a = get_antecedent (α)
          visit_clause ( ω_a , x_α , d, Ψ , Θ )
     }
    back_track (b)
    return SUCCESS
end function
```

**Figura 5.5**: Análise de Conflitos [40]

Na Figura 5.6 são mostrados os detalhes da técnica MVS.

```
function visit_clause ( ω_a , x_α , d, Ψ , Θ )
  for each l_i ∈ ω_a do{
      if (( x_i = get_variable (l_i)) != x_α ) then  {
         x_i .resolve_set = x_i .resolve_set ⋃ l_i.val_set }
         if (decision_level (x_i) = d) then {
             Ψ = Ψ ⋃  { x_i}  % still to be precessed }
         else { % x_i assigned at earlier decision
              Θ = Θ  ⋃ { x_i}  % to be learned }
      else { % x_i is the resolving variable
         if (flag (x_i) ) = PREVIOUSLY_ASSIGNED) then {
          x_i .resolve_set = x_i .resolve_set ⋂ l_i.val_set  }
          if (decision_level (x_i) = d ) then {
             Ψ = Ψ ⋃  { x_i}  % still to be precessed }
         else {  %x_i assigned at earlier decision
              Θ = Θ  ⋃ { x_i}  % to be learned }
      }
   }
end function
```

**Figura 5.6**: Processo de aprendizado MV-SAT [40]

O processo de análise de conflitos é inicializado pela cláusula conflitante. O nível de decisão máximo das variáveis na cláusula é o que determina o *backtrack* (se o nível for igual a 0 significa que o *backtrack* alcançou a raiz). Isto significa que um conflito ocorreu antes que qualquer decisão fosse tomada[4].

Devido à ocorrência do conflito, o algoritmo retorna *FAILURE* e o resolvedor reporta UNSAT (Figura 5.5). Durante a análise de conflito, mantemos $\Theta$ como o conjunto variáveis que serão incluídas na nova cláusula e $\Psi$ o conjunto de variáveis correspondentes a um assinalmento no nível corrente responsável pelo conflito [40][5]. Esses dois conjuntos juntos representam um corte no grafo de implicação (Figura 5.2). Durante cada visita da cláusula antecedente, os dois conjuntos de variáveis são atualizados incluindo o assinalamento de variáveis no nível de decisão $\Theta$ e do nível de decisão $\Psi$. No mesmo momento, o conjunto de valores auxiliares para cada variável é atualizado (Figura 5.6).

As regras de construção para o conjunto de valores auxiliares são baseados em regras de inferência [6]. Primeiramente checamos se a variável associada ao literal visitado é a variável resolvente. Se não for, incluimos o literal no resolvente, e se for, uma operação de intersecção é realizada para derivar um novo conjunto de valores do literal. Observe que na análise de conflitos MV, é bem possível que a intersecção não resulte num conjunto vazio em contraste com o caso

[4]Na atribuição de valores às variáveis ocorreu um conflito.

[5]Muito importante para que numa próxima atribuição de valores não sejam levados em consideração o valores dos literais que já apresentaram conflitos.

[6]Apresentada no Capítulo 4.

binário [40].

Seja dado um assinalamento $x_2 = \{0, 3\}$ e um assinalamento corrente $x_1 = \{1\}$. Isso levará a implicação $x_2 = \{3\}$ por $\omega_1$ (onde $\omega_1 = (x_1^{\{0,2\}} + x_2^{\{1,2,3\}})$). Neste exemplo, a implicação é afetada pelo assinalamento antecedente no qual é gravado por *PREVIOUSLY ASSIGNED* (Figura 5.6). Durante a análise de conflitos quando $\omega_1$ é visitado pelo assinalamento antecedente $x_2 = \{3\}$, o *flag* identifica que o assinalamento foi afetado por um mais recente. Como resultado $x_2$ é adicionado a $\Theta$. O assinalamento mais recente derivado do conjunto $\{1\}$ é construído pela intersecção de $\{1, 2, 3\}$, o literal unitário associado à implicação e $\{0, 1\}$ que é o atual conjunto de valores auxiliares. Observe que a construção do conjunto de valores é a mínima relativo à $\{1, 2\}$ que representa o complemento do assinalamento mais recente.

$$\omega_1 = (x_1^{\{0,2\}} + x_2^{\{1,2,3\}})$$
$$\omega_2 = (x_3^{\{1,4\}} + x_2^{\{0,1\}})$$
$$\overline{\omega_l = (x_1^{\{0,2\}} + x_2^{\{1\}} + x_3^{\{1,4\}})}$$

Uma importante vantagem das técnicas MVS é que o processo de procura baseado no *large decision scheme* não leva necessariamente a um aprendizado fraco. Por exemplo o $\omega_{l1}$ utilizado pode ser aprendido pelas decisões $x_4 = \{0, 2\}$, $x_3 = \{2, 3\}$ que corresponde a um espaço de procura *large* [40].

## 5.5 Resultados

Nesta seção introduzimos a formulação de *benchmarks* para MV SAT usados nos experimentos e avaliamos a eficiência das técnicas MVS. Apresentamos também uma comparação entre a heurística Zchaff e CAMA. Os resultados foram retirados da referência [40].

MVSIS [6] *benchmarks* são derivados de aplicações em *data mining* e inteligência artificial. Para comparar a performance de CAMA com Zchaff, usamos a codificação OHE.

Na Tabela 5.3 é comparado *large and small decision* de CAMA. O resultado sugere que *large decision scheme* trabalha melhor que o *small decision scheme*.

| Classe / Heurística | Small(s) | Large(s) |
|---|---|---|
| m4-6 | 0.84 | 0.35 |
| m4-8 | 6.25 | 2.47 |
| m4-10 | 39.95 | 11.82 |
| hole6 | 0.53 | 0.10 |
| hole7 | 13.05 | 0.43 |

**Tabela 5.3**: *Benchmarks* com duas diferentes heurísticas de decisões de CAMA

Na Tabela 5.4 comparamos dois *learning schemes*. Um usa técnica multivalorada enquanto

que a outra não. A vantagem de se utilizar técnicas multivaloradas é a redução do número de decisões, de cláusulas adicionadas e implicações que resolvem tais instâncias. O tempo de execução aumenta drasticamente quando se resolve instâncias do problema *Pigeon Hole* sem usar técnicas multivaloradas.

| Classe / Heurística | Sem MVS(s) | MVS Technique(s) |
|---|---|---|
| m4-6 | 0.36 | 0.35 |
| m4-8 | 2.50 | 2.47 |
| m4-10 | 12.90 | 11.82 |
| m4-12 | 36.89 | 36.46 |
| hole6 | 0.48 | 0.10 |
| hole7 | 16.35 | 0.43 |

**Tabela 5.4**: Solução de CAMA sem utilizar técnicas multivaloradas para os *Benchmarks*

| Classe / Heurística | Zchaff(s) | CAMA(s) |
|---|---|---|
| m4-6 | 1.00 | 0.35 |
| m4-8 | 7.99 | 2.47 |
| m4-10 | 45.37 | 11.82 |
| m4-12 | 112.09 | 36.46 |
| hole6 | 0.05 | 0.10 |
| hole7 | 1.15 | 0.43 |

**Tabela 5.5**: Tabela Comparativa entre Zchaff e CAMA

Observe que o tempo de execução de CAMA é inferior ao Zchaff nas classes de problemas apresentadas (ou seja, CAMA resolve o problema mais rápido que Zchaff) como mostrado na Tabela 5.5. Sabemos que a heurística CAMA é baseada nos procedimentos desenvolvidos em ZChaff com algumas melhorias que resultaram em ganho no tempo de execução.

## 5.6 Considerações finais

Neste capítulo apresentamos uma primeira heurística utilizando lógica ternária baseada no procedimento de Davis-Putnam utilizando a codificação *double rail*. Apresentamos também uma nova heurística utilizando lógica multivalorada denominada CAMA e avaliamos sua performance tanto comparada com Zchaff binária quanto usando técnicas multivaloradas.

Por ser uma heurística bastante nova e trabalhar com problemas multivalorados o estudo e as aplicações ainda estão sendo desenvolvidas. Observou-se que os avanços apresentados nessa heurística apresentam bons resultados quando comparado à uma heurística consagrada como Zchaff.

# Capítulo 6

# Conclusões

*"O Captain! my Captain! our fearful trip is done,*
*The ship has weather'd every rack,*
*the prize we sought is won,*
*The port is near, the bells I hear, the people all exulting,*
*While follow eyes the steady keel, the vessel grim and daring;*
*But O heart! heart! heart!*
*O the bleeding drops of red,*
*Where on the deck the Captain lies,*
*Fallen cold and dead."*
*Walt Whitman*

## 6.1 Objetivos alcançados

O objetivo dessa dissertação foi confeccionar um texto didático para pessoas com um conhecimento básico na área de ciência da computação que desejam aprender um pouco mais sobre heurísticas para o problema SAT. O texto explica de forma relativamente simples noções iniciais das classes de problemas $P$, $NP$ e $NP$-completos. Definimos formalmente um problema $NP$-completo denominado SAT e explicamos detalhadamente as principais heurísticas utilizadas para a sua solução. Estendemos os principais algoritmos binários para problemas multivalorados, apresentando duas heurísticas importantes que trabalham com conceitos ternários e multivalorados (Davis-Putnam Binário Estendido e CAMA). Fizemos diversas implementações e apresentamos dados empíricos comparativos com o intuito de fornecer ao leitor conclusões acerca de qual heurística utilizar em determinados problemas. Foi verificado que Zchaff e Berkmin apresentam melhores resultados em relação às outras heurísticas apresentadas.

## 6.2 Conclusões

As seguintes conclusões podem ser extraídas da presente dissertação:

1. A presente dissertação analisa os principais conceitos e as principais heurísticas resultando

num guia inicial e básico para se trabalhar o problema SAT. Para conhecer melhor as heurísticas é necessário estudar as referências citadas em cada Capítulo já que a bibliografia é bastante extensa.

2. A utilização de heurísticas, principalmente depois de Davis-Putnam, vêm apresentando resultados expressivos na busca de soluções para problemas SAT. Os resultados apresentados no item 3.9.1 nos mostram que grande parte dos problemas não conseguem ser resolvidos pela heurística Davis-Putnam (DP).

3. A primeira implementação atribuída a Davis-Putnam e Lovelland resultou num novo paradigma para tratar o problema SAT. Apesar de não apresentar bons resultados, a maioria das heurísticas ainda utiliza seu procedimento básico.

4. O avanço nas heurísticas para enfrentar o problema SAT incorpora idéias de algoritmos anteriores. Uma heurística apresenta melhores resultados que outras absorvendo avanços teóricos de procedimentos passados.

5. Não se pode afirmar com precisão que a heurística Berkmin é melhor e mais robusta que a heurística Zchaff. Entretanto pode-se afirmar que ambas estão hoje no topo das mais poderosas na busca por soluções para o problema SAT. Em problemas UNSAT Berkmin apresenta melhores resultados [25].

6. Lógicas multivaloradas vêm se tornando importantes em várias áreas do conhecimento[1]. As formas de representação e codificação apresentadas no Capítulo 4 tem como objetivo utilizar a lógica binária estendida e multivalorada na confecção das heuristicas Davis-Putnam Binário Estendido e CAMA.

7. A heurística CAMA baseada em Zchaff apresenta resultados importantes tanto no tratamento de problemas multivalorados quanto em solução de problemas binários.

8. Quando em uma CNF temos somente literais positivos em uma cláusula, a escolha de um literal da menor cláusula apresenta melhores resultados em problemas *Latin Square*. Este argumento pode ser verificado empiricamente pelas Tabelas do item 3.9.1 e é um resultado muito importante.

## 6.3 Trabalhos Futuros

O objetivo desta dissertação foi o de analisar de forma minuciosa as principais heurísticas para resolução de problemas SAT. A realização de testes empíricos teve como função demonstrar as melhores performances das heurísticas para classes diferentes de problemas. Com base nestes

[1] Algumas delas: Lógica Fuzzy [51], Lógica Paraconsistente [4, 20].

resultados, está sendo desenvolvido uma implementação em *hardware* da heurística CAMA para resolução de problemas binários e multivalorados.

Já que observamos que para problemas onde há uma cláusula positiva e se escolhermos a menor dessas cláusulas verificamos que SATO apresenta melhores resultados e sabendo que Berkmin funciona melhor em fórmulas UNSAT, podemos estudar o comportamento das fórmulas ao longo do processamento da heurística. Já que Zchaff apresenta melhores resultados na média e de posse das informações de SATO e Berkmin, entender um pouco melhor o comportamento das CNFs resultaria uma importante contribuição para o aumento da performance dos algoritmos.

# Bibliografia


[1] M. Ajtai. The complexity of the pigeonhole principle. In *Proceedings of the IEEE 29th Annual Symposium on Foundations of Computer Science*, pages 346–355, Los Alamitos, CA, 1988.

[2] O. Arieli and A. Avron. The value of the four values. *Artificial Intelligence*, 1:97–141, 1998.

[3] P. Baumgartner. A First-Order Logic Davis-Putnam-Logemann-Loveland Procedure. Fachberichte Informatik 3–2002, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2002.

[4] P. Besnard, T. Schaub, H. Tompits, and S. Woltran. Paraconsistent reasoning via quantified boolean formulas, I: Axiomatising signed systems. pages 1–15.

[5] A. Billionnet and A. Sutter. An efficient algorithm for the 3 satisfiability problem. *Operation Research Letters*, 12:29–36, 1992.

[6] R. K. Brayton and S. P. Khatri. Multi-valued logic synthesis. *Proceedings of International Conference on VLSI Design*, January 1999.

[7] F. M. Brown. *The Logic of Boolean Equations*. Kluwer Academir Publishers, 1990.

[8] R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computer Surveys*, 24(3), September 1992.

[9] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[10] T. Castell. *Consistance et déduction en logique propositionnelle*. PhD thesis, Thèse de doctorat de l'Université Paul Sabatier, Toulouse, 1997.

[11] T. Castell and M. Cayrol. Computation of prime implicates and prime implicants by the davis and putnam procedure. In *Proceedings of the ECAI'96 Workshop on Advances in Propositional Deduction*, pages 5–10, Budapest (Hungary), 1996.

[12] T. C. Chang. *Network resource allocation using an expert system with fuzzy logic reasoning*. Phd thesis, University of California at Berkeley, Calif., 1987.

[13] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third IEEE Symposium on the Foundations of Computer Science*, pages 151–158, 1971.

[14] S. A. Cook and D. G. Mitchell. Finding hard instances of the satisfiability problem: A survey. In Du, Gu, and Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–17. American Mathematical Society, 1997.

[15] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, pages 148–159, 1996.

[16] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93)*, pages 21–27, 1993.

[17] M. J. Cresswell and G. E. Hughes. *An Introduction to Modal Logic*. Methuen, London, 1968.

[18] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, 1990.

[19] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[20] H. Decker, J. Villadsen, and T. Waragai, editors. *Paraconsistent Computational Logic*, volume 95 of *Datalogiske Skrifter*, Roskilde, Denmark, July 27 2002. Roskilde University. Possibly updated PCL 2002 papers available at CoRR (Computing Research Repository).

[21] O. Dubois, P. André, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. In D. Johnson and M. Trick, editors, *Second DIMACS implementation challenge : cliques, coloring and satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 415–436. American Mathematical Society, 1996.

[22] J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, Departement of computer and Information science, University of Pennsylvania, Philadelphia, 1995.

[23] I. Gent, H. van Maaren, and T. Walsh, editors. *SAT20000: Highlights of Satisfiability Research in the year 2000*. Frontiers in Artificial Intelligence and Applications. Kluwer Academic, 2000.

[24] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, Mar. 2002.

[25] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, Mar. 2002.

[26] J. Gu, P. Purdom, J. Franco, and B. Wah. Algorithms for the satisfiability (sat) problem: a survey, 1996.

[27] M. Herbstritt. zchaff: Modifications and extensions. Technical Report TR-188, Albert-Ludwigs University, July 2003.

[28] D. R. Hofstadter. *Godel, Escher, Bach: an Eternal Golden Braid*. Basic Books, 1979.

[29] H. H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. In Gent et al. [23], pages 283–292.

[30] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[31] L. Huelsbergen. A representation for dynamic graphs in reconfigurable hardware and its application to fundamental graph algorithms. In *FPGA*, pages 105–115, 2000.

[32] M. Irgens. Constructive heuristics for satisfiability problems. Technical Report TR 97-18, Nov. 1997.

[33] T. S. J. Hooker, H. Hoos. Dimacs phole. In *benchmarks avaible at http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html*, 2000.

[34] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley and Sons, Inc., 1991.

[35] H. Kautz and B. Selman, editors. *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*, volume 9. Elsevier Science Publishers, June 2001. LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001).

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

[37] T. Larrabee. *Efficient Generation of Test Patterns Using Boolean Satisfiability*. PhD thesis, Stanford University, Department of Computer Science, February 1990.

[38] T. Larrabee. Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, January 1992.

[39] C.-M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. pages 366–371.

[40] C. Liu, A. Kuehlmann, and M. W. Moskewicz. Cama: A multi-valued satisfiability solver. In *Proceedings of the 2003 International Conference on Computer-Aided Design (ICCAD'03)*, page 326. IEEE Computer Society, 2003.

[41] I. Lynce, L. Baptista, and J. ao P. Marques Silva. Stochastic systematic search algorithms for satisfiability. In Kautz and Selman [35]. LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001).

[42] I. Lynce and J. Marques-Silva. Efficient data structures for fast sat solvers. In *Technical Report RT 05 2001*, 11 2001.

[43] S. Malik. The quest for efficient Boolean satisfiability solvers. In *CADE:2002*, pages 295–313.

[44] J. P. Marques-Silva and T. Glass. Combinational Equivalence Checking Using Satisfiability and Recursive Learning. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, 1999.

[45] J. P. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[46] J. P. Marques-Silva and K. A. Sakallah. Boolean Satisfiability in Electronic Design Automation. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, June 2000.

[47] W. McCune. A davis-putnam program and its application to finite first-order model search: Quasigroup existence problem. Technical report, Technical Memorandum ANL/MCS-TM-194, 1994.

[48] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distribution of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 440–446, 1992.

[49] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[50] G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar. Satisfiability-based layout revisited: detailed routing of complex fpgas via search-based boolean sat. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 167–175. ACM Press, 1999.

[51] W. Pedrycz. Fuzzy dynamic systems. In D. Driankov, P. W. Eklund, and A. L. Ralescu, editors, *Fuzzy Logic and Fuzzy Control(IJCAI-91)*, pages 37–44. Springer, Berlin, Heidelberg, 1991.

[52] S. V. Campos. *A Quantitative Approach to the Formal Verification of Real-Time Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[53] SATLive. Satlive , up to date links for the satisfiability problem. http://www.haifa.il.ibm.com/projects/verification, 2004.

[54] R. C. Schrag. Compilation of critically constrained knowledge bases. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)*, pages 510–515, 1996.

[55] B. Selman, H. A. Kautz, and D. A. McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 50–54, 1997.

[56] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI–92)*, San Jose, CA, 1992.

[57] J. Silva and K. Sakallah. Conflict analysis in search algorithms for propositional satisfiability, 1996.

[58] J. Silva and K. Sakallah. GRASP – A new search algorithm for satisfiability. Technical Report CSE-TR-292-96, 10 1996.

[59] L. Simon. SAT-Ex. http://www.lri.fr/ simon/satex/satex.php3, 2000.

[60] R. L. R. Thomas H. Cormen, Charles E. Leiserson and C. Stein. *Algoritmos.* Editora Campus, Rua Sete de Setembro 111, 2002.

[61] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42, 1936.

[62] A. M. Turing. Computing machinery and intelligence. *Mind*, 49:433–460, 1950.

[63] R. Zabih and D. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proc. of AAAI-88*, pages 155–160, St. Paul, MN, 1988.

[64] H. Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.

[65] H. Zhang. Specifying latin square problems in propositional logic. *Essays in Honor of Larry Wos,Chapter 6*, 1997.

[66] H. Zhang and M. Stickel. Implementing the davis-putnam algorithm by tries. Technical report, Dept. of Computer Science, University of Iowa, 1994.

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

[68] J. Zhang and H. Zhang. Sem: a system for enumerating models. *IJCAI95*, pages 298–303, 1995.

[69] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, Nov. 2001.