FERNANDA AKL FARIA LASMAR

ANÁLISE DE DESEMPENHO DE ALGORITMOS PARA O PROBLEMA DA SATISFATIBILIDADE BOOLEANA

LAVRAS – MG

2010

FERNANDA AKL FARIA LASMAR

ANÁLISE DE DESEMPENHO DE ALGORITMOS PARA O PROBLEMA DA SATISFATIBILIDADE BOOLEANA

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

Orientador:

Prof. Dr. Joaquim Quinteiro Uchôa

LAVRAS – MG

2010

FERNANDA AKL FARIA LASMAR

ANÁLISE DE DESEMPENHO DE ALGORITMOS PARA O PROBLEMA DA SATISFATIBILIDADE BOOLEANA

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

APROVADA em ____ de _____ de _____

_____
Me. Cristiano Leite de Castro – UFLA

_____
Dr. Bruno de Oliveira Schneider – UFLA

Orientador:

_____
Prof. Dr. Joaquim Quinteiro Uchôa

LAVRAS - MG

2010

A Deus em primeiro lugar.

Ao meu pai Luis Cezar Akl Lasmar.

A minha mãe Sione Faria Lasmar.

A minha irmã Jacqueline Akl Faria Lasmar

Aos meus avôs (in memoriam).

Ao Henrique Ribeiro Rezende.

Aos meus familiares.

DEDICO.

AGRADECIMENTOS

Agradeço a minha família pelo apoio que me deram sempre.

Em especial aos meus pais por terem me ajudado em toda essa caminhada, agradeço por estarem sempre ao meu lado me dando carinho, me aconselhando e me ajudando.

A minha irmã que sempre me ajudou que foi mais do que uma irmã pra mim, foi uma verdadeira amiga.

Aos meus amigos de turma que estiveram sempre unidos, com muita alegria e lealdade, especialmente ao meu namorado que foi sempre companheiro e amigo me ajudando durante todos estes anos.

Agradeço aos professores do Departamento de Ciência Computação da UFLA que ajudaram muito no meu aprendizado durante esses anos.

Agradeço ao Prof. Joaquim Quinteiro Uchôa pela oportunidade e pela dedicação em orientar-me durante este trabalho.

RESUMO

Esta pesquisa tem como principal objetivo investigar abordagens aos problemas da satisfatibilidade booleana, apresentando alguns dos algoritmos que abordam esse problema e verificando quais destes possuem melhor tempo de processamento.

Uma análise comparativa é feita entre os algoritmos Davis-Putnam e DPLL, com o intuito de verificar o tamanho da instância que cada algoritmo consegue solucionar em tempo polinomial.

Palavras-chave: Satisfatibilidade booleana; Davis-Putnam; DPLL.

ABSTRACT

The main objective of this research is to investigate approaches to the problem of Boolean satisfiability, we present some algorithms that address this problem and we looking for ones that have better processing time.

A comparison is made between the Davis-Putnam algorithms and DPLL, in order to check the size of the instance that each algorithm can solve in polynomial time.

Keywords: Boolean satisfiability; Davis-Putnam; DPLL.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

FNC	Forma Normal Conjuntiva
SAT	Satisfatibilidade Boolena
DPLL	Davis-Putnam-Logemann-Loveland
DP	Davis-Putnam

SUMÁRIO

1. INTRODUÇÃO

1.1 Contextualização e Motivação

O problema da Satisfatibilidade Booleana (SAT) consiste em verificar se, há uma atribuição de valores verdade para as variáveis de uma fórmula lógica, que tornem esta fórmula verdadeira. Os algoritmos para resolver o problema SAT irão retornar, se a fórmula é satisfatível ou não, analisando as cláusulas de cada instância.

O problema SAT foi o primeiro problema a ser identificado como pertencendo à classe NP-Completo. Isso significa que ainda não é conhecido nenhum algoritmo determinístico que o resolva em tempo polinomial. Sua solução implicaria na solução em tempo polinomial de vários outros problemas desta mesma classe, como o problema do caixeiro viajante, e significaria o colapso entre as classes P e NP, que é atualmente um dos grandes problemas em aberto da computação.

O problema SAT pertence à classe dos problemas de decisão, isto é, problemas que admitem dois tipos de solução: sim ou não. Uma instância deste problema é dada por um conjunto de cláusulas específicas. O termo instância é um neologismo do inglês que, no contexto, significa um exemplo, uma amostra. Porém, os algoritmos podem ser modificados para retornar quais valores verdade foram associados a cada uma das variáveis. Assim, é possível identificar dois tipos de algoritmos para o problema SAT, os completos e os métodos estocásticos.

Os métodos completos encontram uma solução se existir, ou mostram que a fórmula não pode ser satisfeita. Já os métodos estocásticos encontram respostas para instâncias satisfatíveis. Os algoritmos Davis-Putnam e DPLL são exemplos de algoritmos completos.

O objetivo deste trabalho é investigar as abordagens do problema da satisfatibilidade booleana, bem como os algoritmos que abordam esse

problema, realizando uma análise comparativa entre os algoritmos estudados. Serão estudados os algoritmos completos clássicos para solução do problema SAT, que são os Algoritmos de Davis-Putnam e sua variação DPLL.

Para o desenvolvimento do trabalho será realizado um levantamento bibliográfico sobre o problema SAT, suas abordagens e algoritmos. O estudo e implementação dos algoritmos Davis-Putnam e DPLL serão realizados, para que possa haver uma análise comparativa entre os algoritmos. A análise será feita do ponto de vista quantitativo, verificando-se o tamanho do problema que cada algoritmo consegue solucionar, já que para uma determinada entrada, o problema cresce de forma exponencial.

Espera-se encontrar como resultado, o algoritmo que melhor soluciona o problema da satisfatibilidade booleana com um menor custo computacional.

O algoritmo Davis-Putnam foi criado por Martin Davis e Hilary Putnam em 1960 [1]. Foi um dos primeiros algoritmos na verificação da satisfatibilidade de uma fórmula, quando esta se encontra na forma normal conjuntiva.

O algoritmo DPLL (Davis-Putnam-Logemann-Loveland) é um procedimento muito eficiente e é utilizado como base para outros solucionadores do problema da satisfatibilidade [2].

1.2 Aplicações

A solução do SAT levaria a solução de vários problemas práticos na área da computação, tais como o planejamento automático, a verificação de *software* e o teste de circuitos [3].

Podem ser testados solucionadores do problema da satisfatibilidade para testar as propriedades de algoritmos de criptografia e chaves. Encontrar

uma atribuição de valores para uma fórmula SAT é equivalente a recuperação de uma chave em um ataque criptoanalítico [4].

Outra aplicação para o problema SAT é a Verificação de Equivalência de Circuitos. Este é um dos problemas que pode ser facilmente representado na forma normal conjuntiva e resolvido por solucionadores SAT. O problema consiste em verificar se, para um conjunto de entradas e um conjunto de saídas, em dois circuitos C_A e C_B, para todas as entradas, as saídas dos circuitos são equivalentes [5].

No processo de fabricação, Circuitos Integrados podem estar sujeitos a defeitos. A Geração Automática de Padrões de Teste utiliza o problema da satisfatibilidade boolena para detectar falhas nestes circuitos. Se a atribuição de valores verdades das entradas for verdadeira é porque houve falha no circuito [5].

Os algoritmos SAT são utilizados também na área de Arquitetura de Computadores, onde esses algoritmos são utilizados na alocação de registradores. O maior problema na alocação de registradores é o de associar variáveis em um programa de computador com os registradores do processador, de tal forma que seja minimizado o tempo de execução dos programas [3].

Além das aplicações na área computacional, algoritmos solucionadores do problema SAT têm aplicações também na área médica. Nesta área encontramos análises de DNA modeladas em SAT para o tratamento de doenças. Outra aplicação é no diagnóstico de carcinomas (tumores malignos desenvolvidos a partir de células epiteliais ou glandulares) [4].

2. REFERENCIAL TEÓRICO

O objetivo deste capítulo é apresentar os principais conceitos utilizados neste trabalho. São abordados os conceitos de lógica proposicional, tabela verdade, forma normal conjuntiva e disjuntiva, os problemas P e NP e também uma descrição dos algoritmos utilizados.

2.1 Lógica Proposicional

A história da Lógica tem início com o filósofo grego Aristóteles (384 -322 a.C.), cuja essência era a teoria do silogismo (certa forma de argumento válido). Kant (1724-1804) achava que nada de essencial havia sido feito em lógica depois do grande filósofo grego.

A lógica Booleana inicia-se com George Boole (1815-1864) e Augustus de Morgan (1806-1871). Mas foi com Gotlob Frege (1848-1925) que houve um grande passo no desenvolvimento da lógica moderna em 1879.

A Lógica está preocupada principalmente com dois conceitos: a verdade e a prova. O estudo sobre a atribuição de valores verdade às variáveis lógicas e a avaliação da verdade das fórmulas contribui com a área da lógica denominada Teoria de Modelos. Já a Teoria das Provas, lida com métodos de inferência, que quando aplicado a fórmulas válidas, produzem novas fórmulas válidas. Tanto a Teoria de Modelos quanto a Teoria das Provas têm sido investigados há séculos por filósofos, lingüistas e matemáticos [6].

A lógica proposicional é um formalismo matemático através do qual podemos abstrair a estrutura de um argumento, eliminado a ambigüidade existente na linguagem natural. Esse formalismo é composto por uma linguagem formal e por um conjunto de regras de inferência que nos

permitem analisar um argumento de forma precisa e decidir sobre sua validade [7].

Um argumento é uma seqüência de premissas seguida de uma conclusão. Dizemos que um argumento é válido quando sua conclusão é uma conseqüência necessária de suas premissas [8]. A lógica proposicional é um sistema formal composto por proposições.

Proposições são afirmações que podem assumir valores verdadeiros ou falsos. Por exemplo, "Paris fica na França" é uma proposição verdadeira, enquanto "A lua é de prata" é uma proposição falsa.

A proposição é o elemento básico a partir do qual argumentos são construídos, sendo também o principal objeto de estudo na lógica proposicional. As proposições podem ser classificadas como simples ou compostas, sendo que, proposições simples são aquelas que não contenham nenhuma outra proposição como parte integrante de si. Já as proposições compostas são formadas por combinações de duas ou mais proposições simples através de um elemento de ligação, denominado conectivo lógico.

2.1.1 Sintaxe da Lógica Proposicional

Os símbolos usados na lógica proposicional são as constantes (falso e verdadeiro), os símbolos proposicionais (são letras minúsculas do alfabeto latino, possivelmente indexadas) e os conectivos lógicos são e, ou, negação e implicação.

Seus símbolos são:

- Símbolos proposicionais: p, q, r,... (letras de um alfabeto)
- Conectivos lógicos: e (Λ), ou (V), negação ($\neg$) e implicação ($\rightarrow$).
- Constantes: Verdadeiro (T), Falso ($\perp$).
- Parênteses.

Se α e β forem fórmulas genéricas, então uma fórmula da forma $\neg \alpha$ é uma negação da fórmula α e dizemos que α e $\neg \alpha$ são fórmulas complementares.

Fórmulas da forma $\alpha \Lambda \beta$ e α V β são denominadas, respectivamente, conjunção e disjunção.

Chama-se conjunção de duas proposições p e q, a proposição representada por "p Λ q", cujo valor lógico é a verdadeiro (V) quando ambas as proposições, p e q, são verdadeiras, e a proposição é falsa (F) nas demais situações.

A disjunção de duas proposições, p ou q, é a proposição representada por "p V q", cujo valor lógico é verdadeiro (V) quando ao menos uma das proposições p ou q forem verdadeiras, e a proposição é falsa (F) quando ambas as proposições "p V q" são falsas.

Uma fórmula da forma $\alpha \rightarrow \beta$ é denominada condicional, sendo α o seu antecedente e β o seu conseqüente. O condicional é uma proposição representada por "se p então q" cujo valor lógico é falso (F) quando p é verdadeira e q é falsa, e a proposição é verdadeira (V) nos outros casos.

A ordem de precedência dos conectivos é: ¬, Λ, V, →. Caso uma ordem diferente seja desejada, podemos usar parênteses. Por exemplo, na fórmula ¬ p Λ q, a negação afeta apenas o símbolo proposicional p. Para que a negação afete toda a proposição, ou seja, a conjunção de p e q, a conjunção pode ser escrita da seguinte forma: ¬ (p Λ q).

Podemos usar a lógica proposicional para formalizar um argumento. No processo de formalização, devemos reconhecer as proposições e conectivos que compõem o argumento, de modo que possamos expressá-lo usando fórmulas.

Por exemplo:

(1) Se a seleção brasileira jogar bem, ganha a copa do mundo.

(2) Se a seleção brasileira não jogar bem, a culpa é do técnico da seleção.
(3) Se a seleção brasileira jogar bem, os torcedores fazem festa.
(4) Os torcedores não fazem festa.

Associando cada sentença a um símbolo proposicional temos:

p: “a seleção brasileira jogar bem”
q: “a seleção brasileira ganha à copa”
r: “o técnico da seleção é culpado”
s: “os torcedores fazem festa”

Através dos símbolos proposicionais propostos acima, as seguintes fórmulas podem ser escritas:

(1) $p \rightarrow q$
(2) $\neg p \rightarrow r$
(3) $p \rightarrow s$
(4) $\neg s$

2.1.2 Semântica da Lógica Proposicional

A lógica clássica é apresentada por três princípios que podem ser formulados como segue:

- Princípio da Identidade: Todo objeto é idêntico a si mesmo. Ou seja, a proposição p é igual a ela mesma, p = p.
- Princípio da Contradição: Dadas duas proposições contraditórias (uma é negação da outra), uma delas é falsa. Ou seja, sendo as proposições p e ¬ p, se p for verdadeira, então ¬ p é falsa.

- Princípio do Terceiro Excluído: Dadas duas proposições contraditórias, uma delas é verdadeira. Sendo p e ¬ p proposições contraditórias, se p for falsa, então ¬ p é verdadeira.

Para a representação dos valores (verdadeiro, falso) das proposições é utilizado tabela-verdade. Tabela-verdade é uma tabela matemática usada em lógica para a verificação de uma fórmula. Tabelas-verdades derivam do trabalho de Gottlob Frege, Charles Peirce e outros da década de 1880, e tomaram a forma atual em 1922 através dos trabalhos de Emil Post e Ludwig Wittgenstein [8].

Na Tabela 1 é apresentada a tabela-verdade para os operadores lógicos Λ (conjunção), V (disjunção) e → (implicação) e a Tabela 2, apresenta a tabela verdade para o operador da negação (¬).

Tabela 1 - Tabela Verdade dos operadores binários [8].

p	q	p Λ q	p V q	p → q
V	V	V	V	V
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

Tabela 2 - Tabela Verdade do operador de negação [8].

p	¬ p
V	F
F	V

Embora a tabela-verdade seja um mecanismo bastante simples para verificar a validade de um argumento, dependendo do tamanho da fórmula, sua construção pode ser inviável. De modo geral, se uma fórmula contém n símbolos proposicionais distintos, sua tabela-verdade terá 2^n linhas (uma linha para cada interpretação possível) [8].

Algumas proposições são verdadeiras para todas as interpretações possíveis. Estas proposições são denominadas tautologias. Intuitivamente falando, uma tautologia é uma verdade universal [6].

Por exemplo: p V ¬ p.

Tabela 3 - Tabela Verdade de uma tautologia [8].

p	¬ p	p V ¬ p
V	F	V
F	V	V

Existem também proposições que são falsas para todas as interpretações. Estas proposições são chamadas contradições.

Por exemplo: p Λ ¬ p.

Tabela 4 - Tabela Verdade de uma contradição [8].

p	¬ p	p Λ ¬ p
V	F	F
F	V	F

2.1.3 Equivalência Lógica

Duas fórmulas α e β são equivalentes, $\alpha \equiv \beta$, se suas tabelas-verdade forem idênticas.

Algumas relações de equivalências podem ser demonstradas da seguinte forma:

- Comutatividade: $\alpha \vee \beta \equiv \beta \vee \alpha$; $\alpha \wedge \beta \equiv \beta \wedge \alpha$.
- Associação: $(\alpha \vee \beta) \vee \gamma \equiv \alpha \vee (\beta \vee \gamma)$; $(\alpha \wedge \beta) \wedge \gamma \equiv \alpha \wedge (\beta \wedge \gamma)$.
- Idempotentes: $\alpha \vee \alpha \equiv \alpha$; $\alpha \wedge \alpha \equiv \alpha$.
- Negação: $\neg\neg \alpha \equiv \alpha$.
- Propriedades de V: $\alpha \vee V \equiv V$; $\alpha \wedge V \equiv \alpha$.
- Propriedades de F: $\alpha \vee F \equiv \alpha$; $\alpha \wedge F \equiv F$.
- Absorção: $\alpha \vee (\alpha \wedge \gamma) \equiv \alpha$; $\alpha \wedge (\alpha \vee \gamma) \equiv \alpha$.
- Distributiva: $\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$; $\alpha \wedge (\beta \vee \gamma) \equiv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$.
- Leis de Morgan: $\neg (\alpha \vee \beta) \equiv \neg \alpha \wedge \neg \beta$; $\neg (\alpha \wedge \beta) \equiv \neg \alpha \vee \neg \beta$.
- Implicação: $\alpha \rightarrow \beta \equiv \neg \alpha \vee \beta$; $\alpha \rightarrow \beta \equiv \neg (\alpha \wedge \neg\beta)$.

2.1.4 Regras de Inferência

Uma regra de inferência é um padrão que estabelece como uma nova fórmula pode ser gerada a partir de outras duas. As regras de inferência representam formas de raciocínio dedutivo [8].

São regras de inferência clássicas:

- Modus Ponens:

$$\frac{\alpha \rightarrow \beta \quad \alpha}{\beta}$$

- Modus Tollens:

$$\frac{\alpha \rightarrow \beta \quad \neg \alpha}{\beta}$$

- Silogismo Hipotético:

$$\frac{\alpha \rightarrow \beta \text{ e } \beta \rightarrow \gamma}{\alpha \rightarrow \gamma}$$

2.2 Formas Normais Canônicas

Uma fórmula lógica pode ser representada em uma forma padrão bem definida, ou seja, em uma forma normal canônica [9].

Uma fórmula denota uma função, que envolve variáveis booleanas. Algumas definições são apresentadas abaixo:

- Um literal é definido como uma variável proposicional, ou a sua negação, respectivamente x e ¬ x. Se o literal x for verdadeiro, conseqüentemente ¬ x é falso.
- Se a variável proposicional não tiver valor atribuído, ela será considerada uma variável livre.
- Uma cláusula é dita vazia se não possuir nenhum literal.
- Uma cláusula que contenha apenas um literal é denominada cláusula unitária.
- Uma cláusula é dita satisfeita se as atribuições já feitas fizeram com que, a resposta do conjunto de literais tem resultado verdadeiro.
- Uma cláusula é dita não resolvida se possuir variável livre ou ainda não for satisfeita.

2.2.1 Forma Normal Conjuntiva (FNC)

A forma normal conjuntiva é uma conjunção de cláusulas.

$$C_1 \Lambda C_2 \Lambda C_3 \Lambda ... \Lambda C_k$$

Exemplo: $(X_1 \text{ V } X_2 \text{ V } X_3 \text{ V } X_4) \Lambda (X_2 \text{ V } X_3)$, onde $(X_1 \text{ V } X_2 \text{ V } X_3 \text{ V } X_4)$ representa uma cláusula da fórmula e $(X_2 \text{ V } X_3)$ representa a outra cláusula.

Tais que essas cláusulas são disjunções de literais

$$L_1 \text{ V } L_2 \text{ V } L_3 \text{ V} ... \text{ V } L_M$$

Exemplo: $(X_2 \text{ V } X_3 \text{ V } X_4)$, formam uma cláusula, onde, X_2, X_3 e X_4 são os literais.

2.2.2 Forma Normal Disjuntiva (FND)

A forma normal disjuntiva é uma disjunção de cláusulas.

$$C_1 \text{ V } C_2 \text{ V } C_3 \text{ V} ... \text{ V } C_k$$

Tais que estas cláusulas são conjunções de literais:

$$L_1 \Lambda L_2 \Lambda L_3 \Lambda ... \Lambda L_M$$

2.3 Problemas P, NP, NP-COMPLETOS

A complexidade de um problema é dada pelo consumo de tempo ou memória de um algoritmo ótimo, que resolva este problema. Um problema é polinomial se existir um algoritmo determinístico que resolva este problema em tempo polinomial. Um problema é não-polinomial se não existir algoritmo polinomial que o resolva. Problemas não-polinomiais são considerados computacionalmente intratáveis.

Algoritmos que para entradas de tamanho n, possuem tempo de execução, no seu pior caso, O (n_k) são chamados algoritmos de tempo polinomial.

Problemas de decisão são problemas que podem ser respondidos com sim ou não.

Um problema de decisão p é um elemento da classe P, se e somente se ele puder ser resolvido por um algoritmo determinístico em tempo polinomial.

Problemas pertencentes à classe NP são problemas de decisão, que podem ser resolvidos em tempo polinomial, usando algoritmos não determinísticos.

Algoritmos não-determinísticos utilizam uma função “escolha”, que faz com que seja escolhido o valor desejado de um conjunto.

Um problema de decisão p está na classe NP, se e somente se, ele puder ser resolvido em tempo polinomial por um algoritmo não-determinístico.

A classe NP-Completo, contém problemas de decisão que podem ser redutíveis polinomialmente a outros problemas da classe NP-Completo.

A classe de problemas NP-completo possui os problemas mais difíceis em NP.

Um problema p é chamado NP-Completo se p ϵ NP, e
p ϵ a classe NP-Difícil.

A redutibilidade polinomial foi revelada por Stephen Cook em 1971, através do Teorema de Cook [10], que prova que o problema da satisfatibilidade booleana é NP-completo e diz que todos os problemas em NP, são redutíveis em tempo polinomial ao SAT [11].

A Figura 1 mostra o Diagrama de Euler para o conjunto de problemas P, NP,NP-completo, e NP-hard.

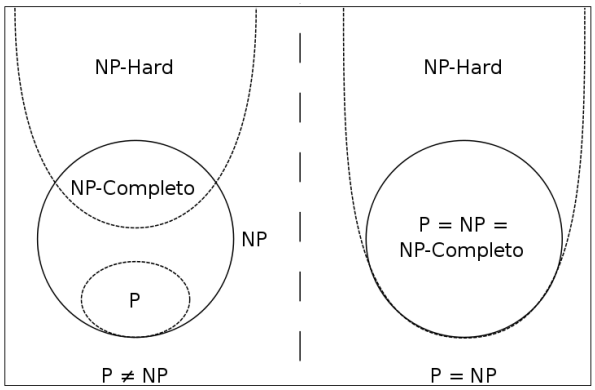

Figura1 - Diagrama de Euler.

2.4 Problema da Satisfatibilidade Boolena

O problema da satisfatibilidade booleana é um problema NP-Completo, e consiste em verificar se existe uma atribuição de valores verdades, para um conjunto de variáveis de uma fórmula booleana, que satisfaça a fórmula. Ou seja, se para um conjunto de variáveis que estejam na forma normal conjuntiva (FNC), existe pelo menos uma atribuição de valores a essas variáveis que tornem a fórmula verdadeira. Se as variáveis estiverem na forma normal disjuntiva (FND) o problema deixa de ser NP-Completo e é facilmente resolvido, basta apenas um termo da fórmula verdadeiro, para que toda a fórmula seja verdadeira, sendo assim satisfeita.

Como SAT pertence à classe dos problemas de decisão, então os algoritmos que o resolvem, ao receberem uma instância do problema,

deverão responder “sim” se o conjunto de cláusulas da instância for satisfatível, ou responder “não” se o conjunto de cláusulas for insatisfatível.

Os algoritmos para resolver a satisfatibilidade booleana podem ser divididos em dois grupos, algoritmos baseado em métodos completos, e os algoritmos baseados em métodos estocásticos.

Dentro da classe dos problemas SAT, existe os problemas 2-SAT que podem ser resolvidos em tempo polinomial e os problemas 3-SAT que não podem ser resolvidos em tempo polinomial.

O problema 3-SAT possui uma fase de transição, que é conhecida como ponto de *crossover*, essa fase de transição é responsável por separar o problema em três tipos, os problemas que são facilmente resolvidos, pois são compostos por poucas cláusulas e poucos símbolos, os que são resolvidos, mas demandam maior custo computacional e os problemas que são resolvidos em tempo exponencial, esta última fase se encontra próxima ao ponto de *crossover*. O *crossover* ocorre em uma razão de aproximadamente 4,35, essa razão é calculada dividindo o número de cláusulas pelo número de variáveis da fórmula. Sabe–se que 50% dos problemas da região de *crossover* são satisfatíveis [12].

Aparentemente problemas com poucas cláusulas parecem fáceis de serem resolvidos, pois são capazes de gerar várias soluções. Problemas com um grande número de cláusulas, também parecem ser facilmente resolvidos, pois um algoritmo inteligente será capaz de resolver rapidamente a maioria ou todas as cláusulas na árvore de busca. Porém problemas que possuem poucas soluções, mas com várias soluções parciais são os mais difíceis de serem resolvidos.

A Figura 2 mostra o efeito do ponto *crossover*. O eixo das ordenadas mostra a porcentagem da satisfatibilidade da fórmula, já o eixo das abscissas apresenta a razão entre o número de variáveis e o número de cláusulas de uma fórmula boolena.

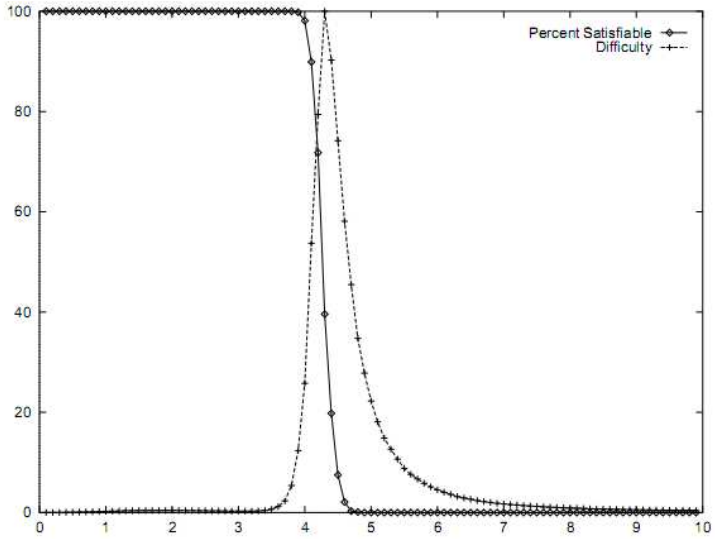

Figura 2 - Efeito do ponto *crossover* [12].

A Figura 3 mostra a relação entre a porcentagem da satisfatibilidade do número de variáveis em função das cláusulas. Cada linha no gráfico mostra o comportamento da fórmula quanto a sua satisfatibilidade através da variação no número de variáveis para um número fixo de cláusulas. O eixo das ordenadas mostra a porcentagem da satisfatibilidade da fórmula, já o eixo das abscissas apresenta a razão entre o número de variáveis e o número de cláusulas de uma fórmula boolena.

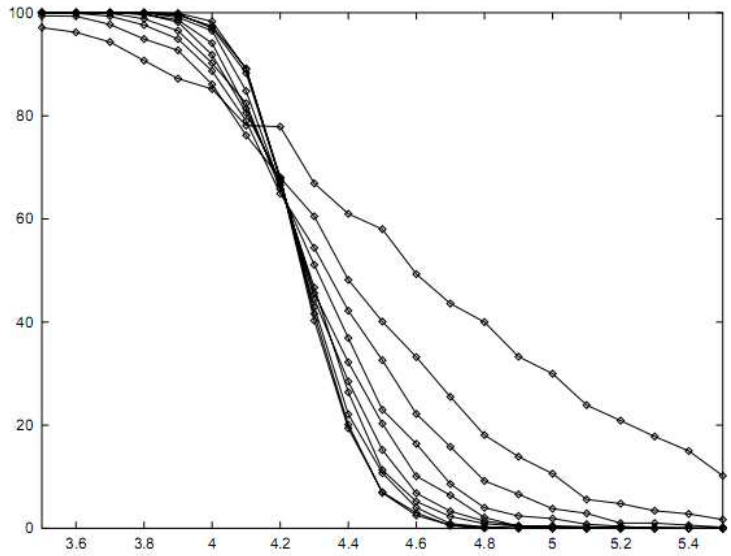

Figura 3 - Porcentagem da satisfatibilidade [12].

2.5 Algoritmos de Resolução de Problemas SAT

O uso de solucionadores para o problema SAT tem extrema importância, pois cada instância de um problema NP-completo pode ser reduzido a uma instância do problema SAT em tempo polinomial.

Como dito anteriormente os algoritmos solucionadores do SAT são divididos em dois grupos, os algoritmos estocástico e os algoritmos completos. Os algoritmos Davis-Putnam e DPLL são algoritmos completos, ou seja, estes algoritmos sempre irão encontrar uma atribuição de valores para variáveis de uma instância se ela for satisfatível, ou dizer que esta instância é insatisfatível, isto se o tempo e o espaço de memória forem suficientes.

Algoritmos estocásticos não podem provar se uma instância não é satisfatível, mas dada uma instância satisfatível, este método é capaz de encontrar uma atribuição de valores para esta instância. Os algoritmos estocásticos são baseados em otimização matemática.

A principal idéia desses algoritmos é a de considerar uma fórmula que tenha cláusulas não satisfeitas como uma função objetivo, e minimizar esse objetivo dando valores para as variáveis.

Algoritmos genéticos, redes neurais e busca local, são métodos que foram utilizados em solucionadores SAT, porém dentre estes métodos os algoritmos baseados em busca local mostraram melhor desempenho [13].

Métodos estocásticos são utilizados em problemas de Inteligência Artificial. Métodos completos são necessários em problemas de verificação e para demonstração se um determinado problema tem solução.

Em 1960 Davis e Putnam publicaram um algoritmo para verificar se uma fórmula proposicional na FNC é não satisfatível.

Em 1962, Davis, Logemann e Loveland publicaram outro algoritmo bastante conhecido e utilizado até os dias atuais, o qual é apenas uma otimização do algoritmo de 1960.

2.5.1 Algoritmo Davis-Putnam

O algoritmo Davis-Putnam foi criado por Martin Davis e Hilary Putnam e utiliza o Algoritmo de *backtracking* para reduzir o espaço de busca. Este algoritmo, para a maioria dos casos, apresenta bom resultado, mas no pior caso ele é exponencial.

O algoritmo é recursivo e cria uma árvore de busca, atribuindo valores verdade e dividindo o problema em problemas menores (Divisão e Conquista).

O algoritmo básico escolhe primeiramente um dos literais da fórmula booleana, assinala um valor para o mesmo, simplifica a fórmula e verifica recursivamente se a fórmula é satisfatível. Se a resposta for sim, então o problema está resolvido. Caso contrário, a mesma verificação recursiva é feita assinalando-se o valor inverso para a variável escolhida inicialmente.

Um dos problemas do algoritmo Davis-Putnam é que se o número de variáveis aumenta pode haver um crescimento exponencial da fórmula, resultando assim em um consumo explosivo de memória. Por esta razão uma variação deste algoritmo foi proposta por Davis, Logemann e Loveland [14].

O algoritmo Davis-Putnam é apresentado abaixo, considerando a fórmula φ na forma normal conjuntiva a saída será satisfatível ou insatisfatível:

1. Se φ possuir uma cláusula vazia, retorne insatisfatível
2. Remova de φ todas as cláusulas que possuam literais complementares. Se φ for vazia retorne satisfatível.
3. Se φ tiver uma cláusula unitária (x) e outra (¬x), retorne insatisfatível
4. Se φ tiver um cláusula unitária (x), remova da fórmula φ todas as cláusulas que contenha x ou seu complemento. Se φ for vazia, retorne satisfatível, senão, se φ tiver uma cláusula unitária, retorne ao passo 1.
5. Enquanto existir uma cláusula unitária, remova todas as cláusulas que tiver o literal presente na cláusula unitária. Se φ for vazia, retorne satisfatível.
6. Selecione uma variável x. Seja C_1 a conjunção de todas as cláusulas que contenham x, e seja C_2 a conjunção das cláusulas que contenham ¬x. Seja C_3 a conjunção das cláusulas que não tenham nem x nem ¬x. Então remova x de C_1 produzindo C_1', e remova ¬x de C_2 produzindo C_2'. Substituindo φ por $(C_1' \vee C_2') \wedge C_3$. Use a propriedade distributiva para transformar φ em uma conjunção de cláusulas. Retorne ao passo 2.

Um exemplo da execução do algoritmo pode ser visto, observando-se a seguinte fórmula: $(\neg X_1) \wedge (X_1 \vee X_2 \vee X_3) \wedge (\neg X_1 \vee X_3)$.

Assumindo a regra de cláusulas unitárias, iremos considerar ¬ X_1 verdadeiro, com isso a fórmula ficará (X_1 V X_2 V X_3) Λ (¬ X_1 V X_3).

Observando que a cláusula (¬X_1 V X_3) contém ¬ X_1 que é verdadeiro, temos que a cláusula também é verdadeira. Então temos a seguinte simplificação da fórmula (X_1 V X_2 V X_3). Sabemos que o literal ¬¬X_1 é falso, então podemos descartá-lo da cláusula, (X_2 V X_3), se considerarmos X_2 ou X_3 como verdadeiro, então a fórmula é satisfatível.

A Figura 4 mostra o desenvolvimento do algoritmo para a fórmula citada acima.

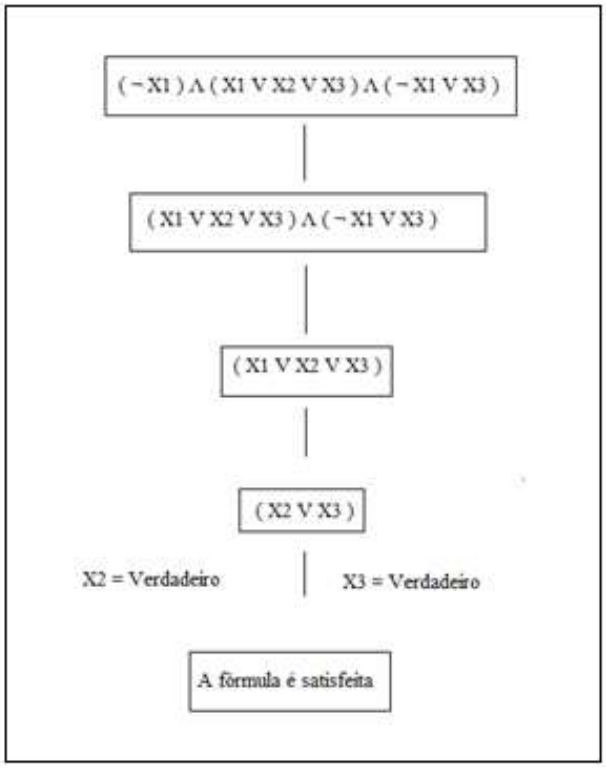

Figura 4 - Execução do exemplo

2.5.2 Algoritmo Davis-Putnam-Logemann-Loveland

A maioria dos algoritmos de resolução para problemas SAT tem origem no algoritmo DPLL, ou Davis-Putnam-Logemann-Loveland, e sempre tentam aperfeiçoar algum aspecto do algoritmo original. O DPLL pode ser definido como um Algoritmo de *backtracking* para decidir a satisfatibilidade de uma fórmula booleana, que pode ser representada em FNC. Este algoritmo foi apresentado em 1962, e é um refinamento do algoritmo Davis-Putnam, que foi desenvolvido em 1960.

O DPLL introduziu basicamente duas melhorias no algoritmo. A primeira delas é chamada de propagação unitária e afirma que se uma cláusula é unitária, ou seja, contém apenas um literal não assinalado, ela pode ser satisfeita assinalando-se o valor desejado a este literal. Na prática, isto acaba diminuindo o espaço de busca. A outra melhoria é chamada de eliminação de literal puro, e afirma que se uma variável ocorrer apenas com uma polaridade na fórmula, ela é chamada de pura. Literais puros podem sempre ser assinalados de uma maneira que tornem todas as cláusulas que os contêm verdadeiras. Dessa maneira, estas cláusulas podem ser apagadas, deixando de fazer parte da busca.

Existe hoje em dia, uma série de variações do algoritmo, cada uma com uma heurística diferente para a escolha do literal que será assinalado primeiro. A eficiência do algoritmo está diretamente ligada a esta escolha.

Os trabalhos atuais geralmente buscam melhorias em três pontos principais do algoritmo original: definir diferentes políticas para a escolha dos literais, definirem novas estruturas de dados para tornar o algoritmo mais rápido, especialmente na parte da propagação de cláusulas unitárias e definindo variantes do algoritmo básico de *backtracking* [5].

O algoritmo DPLL genérico é apresentado abaixo, considerando a fórmula φ na forma normal conjuntiva a saída será satisfatível ou insatisfatível:

1. Se não existir nenhuma cláusula unitária em φ, vá para o passo 3.
2. Se houver uma cláusula unitária (x) em φ, remova de φ, todas as cláusulas que tiverem o literal x ou o seu complemento ¬x. Retorne ao passo 1.
3. Se φ tiver uma cláusula vazia, retorne insatisfatível. Senão, enquanto houver um literal na fórmula, retire todas as cláusulas que este literal esteja. Se então houver uma cláusula vazia, retorne satisfatível.
4. Selecione uma variável qualquer x.
5. Execute recursivamente o algoritmo para $\varphi \Lambda$ (x). Se a chamada recursiva retornar satisfatível, então retorne satisfatível.
6. Execute recursivamente o algoritmo para $\varphi \Lambda$ (¬x). Se a chamada recursiva retornar satisfatível, retorne então satisfatível.

Na Figura 5 é apresentado o pseudo-código do algoritmo DPLL, a palavra BCP apresentada na figura se refere ao algoritmo de *backtracking.*

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

Figura 5 - Pseudo-código DPLL [14].

A execução do algoritmo pode ser descrita da seguinte forma, seja a fórmula na FNC: $(X_1 \text{ V } X_2 \text{ V } X_3) \Lambda (\neg X_1 \text{ V } X_2) \Lambda (\neg X_2 \text{ V } X_3) \Lambda (\neg X_1 \text{ V } \neg X_2 \text{ V } \neg X_3)$.

Se considerarmos X_1 verdadeiro teremos então uma nova fórmula, $(X_2) \Lambda (\neg X_2 \text{ V } X_3) \Lambda (\neg X_2 \text{ V } \neg X_3)$. Analisando esta fórmula, se considerarmos X_2 verdadeiro teríamos a fórmula igual a $(X_3) \Lambda (\neg X_3)$.

Como o resultado desta operação terá como valor verdade falso, iremos considerar $\neg X_2$ verdadeiro, então a fórmula seria verdadeira, pois o resultado desta operação, $(\neg X_2) \Lambda (\neg X_2)$ é verdadeiro. Sendo assim a fórmula pode ser satisfeita.

Se no primeiro passo a escolha fosse por $\neg X_1$ verdadeiro, a fórmula então ficaria do seguinte modo: $(X_2 \text{ V } X_3) \Lambda (\neg X_2 \text{ V } X_3)$. Fazendo $\neg X_3$ verdadeiro, a fórmula ficaria falsa, pois (X_2) V $(\neg X_2)$ é falso.

Mas se escolhermos X_3 verdadeiro então a fórmula seria satisfazível, pois o resultado das operações na fórmula seria verdadeiro.

A Figura 6 mostra o que foi descrito acima:

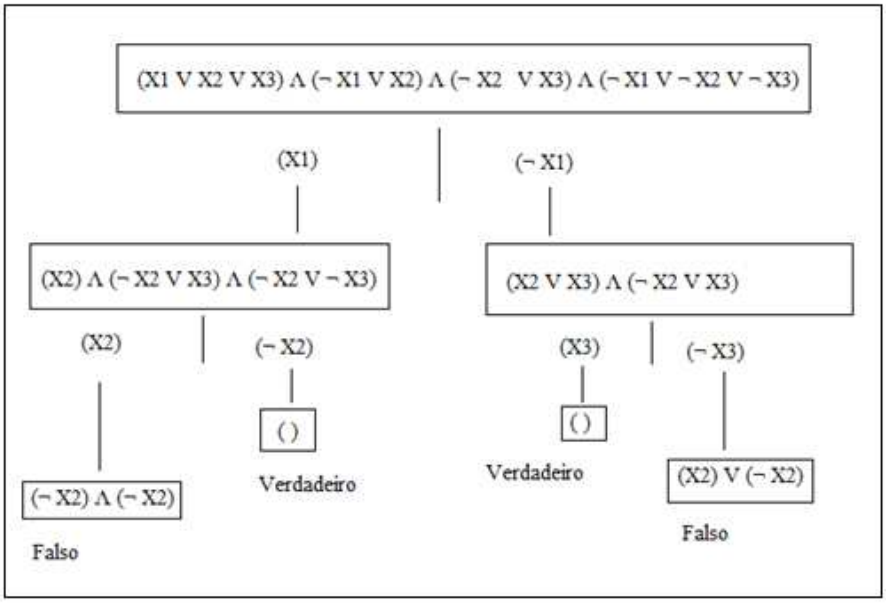

Figura 6 - Execução do exemplo.

3. METODOLOGIA

A ciência tem como meta principal abordar a veracidade dos fatos, e para que um conhecimento seja considerado como científico, devemos utilizar métodos já aceitos e empregados pela comunidade científica. Portanto, para atingirmos o conhecimento, é imprescindível a elaboração de métodos científicos baseados em um conjunto de procedimentos intelectuais e técnicos previamente adotados.

A pesquisa metodológica está relacionada a estudos que visam desenvolver ferramentas para a captação ou manipulação de dados relativos à nossa realidade. Portanto, ela está associada a caminhos, formas, métodos, e procedimentos para alcançarmos um objetivo proposto [15].

Para o desenvolvimento deste trabalho foi feito um levantamento bibliográfico sobre o problema da satisfatibilidade booleana, a maneira como sua solução traria vantagens e variadas descobertas no campo computacional, o modo como este problema vem sendo abordado e os algoritmos utilizados para melhorar o tempo de processamento que no pior caso tende a ser exponencial. A análise de algoritmos tem como foco o algoritmo clássico Davis-Putnam que vem sendo utilizado como base para muitos outros solucionadores do problema. Dados sobre este algoritmo e algumas de suas variações foram levantados para uma melhor descrição do processo.

O principal foco deste trabalho é o estudo e a implementação destes algoritmos, para ser feito uma análise detalhada do desempenho de cada um destes, lembrando que para todos estes algoritmos analisados o seu pior caso é sempre exponencial.

A análise comparativa dos algoritmos foi feita abordando como métrica a análise quantitativa para verificar o comportamento dos algoritmos para solucionar problemas com variados tamanhos de instâncias, já que para uma entrada o problema cresce de forma exponencial.

Quanto aos procedimentos técnicos, ou meios de investigação, o trabalho se enquadra melhor à pesquisa experimental, por seu caráter exato e por envolver muitas variáveis que podem ser medidas e quantificadas. Sabe-se hoje em dia que as vantagens da pesquisa experimental são enormes, pois é a partir dela que conseguimos obter os maiores avanços nas ciências físicas e biológicas. Ela nos permite obter uma maior nitidez, exatidão, e objetividade nos resultados, porém, exige um controle extremo das variáveis envolvidas.

Devido à natureza este trabalho se enquadra como pesquisa aplicada, pois tem como finalidade de responder as questões iniciais, este trabalho é voltado para o desenvolvimento científico.

Neste trabalho foi utilizado o *benchmark* SATLIB [16]. Geralmente *benchmarks* devem oferecer uma ampla variedade de situações diferentes para serem usadas para avaliar diferentes tipos de algoritmos e ser ao máximo, imparciais.

O SATLIB oferece quatro tipos diferentes de problemas, e apresenta um conjunto de testes com instâncias insatisfatíveis e satisfatíveis, incluindo amostragens de problemas 3-SAT, que fazem parte da região do ponto de *crossover*. Os arquivos utilizados para avaliar os algoritmos pertencem, a classe de problemas 3-SAT.

Um exemplo que contém uma amostra da disposição dos dados do arquivo de entrada é apresentado na Figura 7.

```
c This Formular is generated by mcnf
c
c    horn? no
c    forced? no
c    mixed sat? no
c    clause length = 3
c
p cnf 20  91
4 -18 19 0
3 18 -5 0
-5 -8 -15 0
-20 7 -16 0
10 -13 -7 0
-12 -9 17 0
17 19 5 0
-16 9 15 0
11 -5 -14 0
18 -10 13 0
-3 11 12 0
-6 -17 -8 0
-18 14 1 0
-19 -15 10 0
12 18 -19 0
-8 4 7 0
-8 -9 4 0
7 17 -15 0
12 -7 -14 0
-10 -11 8 0
2 -15 -11 0
9 6 1 0
-11 20 -17 0
9 -15 13 0
12 -7 -17 0
-18 -2 20 0
```

Figura 7 - Amostra do arquivo SATLIB.

Cada linha do arquivo corresponde a uma cláusula, e cada número corresponde a uma variável. A negação da variável é representada pelo sinal negativo na frente do número.

A linguagem de programação utilizada foi Python, utilizando o ambiente de programação Python 2.7 para Windows.

O algoritmo foi executado em duas máquinas com configurações distintas. A máquina A possui sistema operacional Windows Vista, com processador de 2.00 GHz e memória RAM de 1 GB. A máquina B possui sistema operacional Windows XP, com processador de 1.60 GHz e memória RAM de 248 MB.

A Figura 8 apresenta um trecho do código do Algoritmo Davis-Putnam.

```
from sets import Set

def Tclauses(formula, vars):
    clausula = Set()
    for subset in formula:
        for var in vars:
            if (var in subset) and (("~" + var) in subset):
                clausula.add(subset)
                break
    return clausula

def satisfiable(formula, vars):
    atribuicao = formula
    for var in vars:
        atribuicao = atribuicao.difference(Tclauses(atribuicao, vars))
        T = Set([formula for formula in atribuicao if var in formula or "~" + var in formula])
        clausula = Set()
        variavel = Set([x for x in T if var in x])
        variavelneg = Set([x for x in T if "~" + var in x])
        for s1 in variavel:
            for s2 in variavelneg:
                clausula = s1.difference(var).union(s2.difference(Set(["~" + var])))
                if not clausula:
                    return False
                clausula.add(clausula)
        atribuicao = atribuicao.difference(T).union(clausula)
        if not atribuicao:
            return True
```

Figura 8 - Amostra código algoritmo Davis-Putnam.

A Figura 9 apresenta um pequeno trecho do código do Algoritmo DPLL.

```
def dpll(clausulas, variaveis, testadas):
    clausulas_des = []
    for c in clausulas:
        atrib =  pl_true(c, testadas)
        if atrib == False:
            return False
        if atrib != True:
            clausulas_des.append(c)
    if not clausulas_des:
        return testadas
    P, value = literal_puro(variaveis, clausulas_des)
    if P:
        return dpll(clausulas, removeall(P, variaveis), extend(testadas, P, value))
    P, value = clausula_unitaria(clausulas, testadas)
    if P:
        return dpll(clausulas, removeall(P, variaveis), extend(testadas, P, value))
    P = variaveis.pop()
    return (dpll(clausulas, variaveis, extend(testadas, P, True)) or
            dpll(clausulas, variaveis, extend(testadas, P, False)))

def literal_puro(variaveis, clausulas_des):
    for s in variaveis:
        var, varneg = False, False
        for c in clausulas_des:
            if not var and s in disjuncts(c):
                var = True
            if not varneg and ~s in disjuncts(c):
                varneg = True
        if var != varneg:
            return s, var
    return None, None

def clausula_unitaria(clausulas, testadas):
    for clause in clausulas:
        aux = 0
        for literal in disjuncts(clause):
            sym = literal_symbol(literal)
            if sym not in testadas:
                aux += 1
                P, value = sym, (literal.op != '~')
        if aux == 1:
            return P, value
    return None, None
```

Figura 9 - Amostra código algoritmo DPLL.

4. RESULTADOS

Após a implementação dos algoritmos Davis-Putnam e DPLL, foram obtidas as seguintes análises de tempos de execução de cada algoritmo em diferentes situações e diferentes máquinas.

As tabelas apresentam as informações da seguinte maneira:

- Número de Cláusulas: há o aumento no número de cláusulas de cada instância, e conseqüentemente o aumento no tempo de execução de cada algoritmo.
- Número de Variáveis: além do aumento no número de cláusulas, há também um aumento no número de variáveis de cada instância, como podemos perceber a cada nova linha de cada algoritmo.
- Tempo de execução: apresenta o cálculo do tempo de execução em segundos de cada algoritmo diante das diversas situações. O tempo de execução é obtido através da média de quatro execuções do mesmo algoritmo para a mesma entrada.

A Tabela 5 apresenta os tempos de execução na máquina A, em que, todas as entradas testadas são satisfatíveis.

Tabela 5 - Resultados obtidos para entradas Satisfatíveis

Algoritmos	Número de Clausulas	Número de Variáveis	Tempo de Execução
Davis-Putnam	218	50	1, 19569882 s
DPLL	218	50	0, 26499996 s
Davis-Putnam	325	75	3, 71705560 s
DPLL	325	75	0, 38799986 s
Davis-Putnam	430	100	7, 30299997 s
DPLL	430	100	0, 77000093 s
Davis-Putnam	538	125	10, 18941000 s
DPLL	538	125	2, 62599988 s
Davis-Putnam	645	150	18, 23659900 s
DPLL	645	150	7, 08699989 s
Davis-Putnam	753	175	339, 37500000 s
DPLL	753	175	48, 58722200 s
Davis-Putnam	860	200	650, 79856501 s
DPLL	860	200	149, 25555696 s
Davis-Putnam	1065	250	1179, 56999025 s
DPLL	1065	250	687, 23658971 s

Através da tabela, podemos perceber que há uma variação no tempo de execução dos algoritmos, e através dessa variação podemos notar que o algoritmo DPLL é mais rápido, executando a mesma instância, do que o algoritmo Davis-Putnam.

A Tabela 6 apresenta os tempos de execução na máquina A, cujas entradas são insatisfatíveis.

Tabela 6 - Resultados obtidos para entradas Insatisfatíveis.

Algoritmos	Número de Clausulas	Número de Variáveis	Tempo de Execução
Davis-Putnam	218	50	2, 19700002 s
DPLL	218	50	0, 45999999 s
Davis-Putnam	325	75	5, 70600008 s
DPLL	325	75	0, 55733331 s
Davis-Putnam	430	100	9, 30299997 s
DPLL	430	100	1, 036666631 s
Davis-Putnam	538	125	13, 88100004 s
DPLL	538	125	4, 291666666 s
Davis-Putnam	645	150	22, 86599993 s
DPLL	645	150	8, 140666643 s
Davis-Putnam	753	175	944, 56898741 s
DPLL	753	175	102, 67700004 s
Davis-Putnam	860	200	1630, 37500000 s
DPLL	860	200	652, 64099979 s
Davis-Putnam	1065	250	3755, 23600000 s
DPLL	1065	250	1902, 15700006 s

As Tabelas 7 e 8 apresentam os resultados obtidos através da execução dos algoritmos na máquina B.

Tabela 7 - Resultado para dados Satisfatíveis.

Algoritmos	Número de Clausulas	Número de Variáveis	Tempo de Execução
Davis-Putnam	218	50	1, 46900001 s
DPLL	218	50	0, 29009657 s
Davis-Putnam	325	75	4, 255589605 s
DPLL	325	75	0, 55000123 s
Davis-Putnam	430	100	8, 91350001 s
DPLL	430	100	0, 87582311 s
Davis-Putnam	538	125	11, 42879555 s
DPLL	538	125	3, 86999987 s
Davis-Putnam	645	150	21, 32589456 s
DPLL	645	150	8, 12500000 s
Davis-Putnam	753	175	530, 95766663 s
DPLL	753	175	51, 80099965 s
Davis-Putnam	860	200	749, 44458452 s
DPLL	860	200	281, 98562156 s
Davis-Putnam	1065	250	1718, 05265789 s
DPLL	1065	250	975, 23652856 s

Tabela 8 - Resultados obtidos para entradas Insatisfatíveis.

Algoritmos	Número de Clausulas	Número de Variáveis	Tempo de Execução
Davis-Putnam	218	50	3, 43799996 s
DPLL	218	50	0, 35899996 s
Davis-Putnam	325	75	7, 70300006 s
DPLL	325	75	0, 70300006 s
Davis-Putnam	430	100	11, 59399986 s
DPLL	430	100	1, 73499989 s
Davis-Putnam	538	125	15, 96899986 s
DPLL	538	125	4, 43700003 s
Davis-Putnam	645	150	25, 15600013 s
DPLL	645	150	10, 96799993 s
Davis-Putnam	753	175	966, 04700017 s
DPLL	753	175	84, 25000000 s
Davis-Putnam	860	200	2096, 85899997 s
DPLL	860	200	794, 70300006 s
Davis-Putnam	1065	250	3805, 70300007 s
DPLL	1065	250	2961, 15599990 s

Os resultados das tabelas acima podem ser analisados através dos gráficos.

O primeiro gráfico mostra o desempenho dos algoritmos diante de instâncias satisfatíveis e insatisfatíveis na Máquinas A.

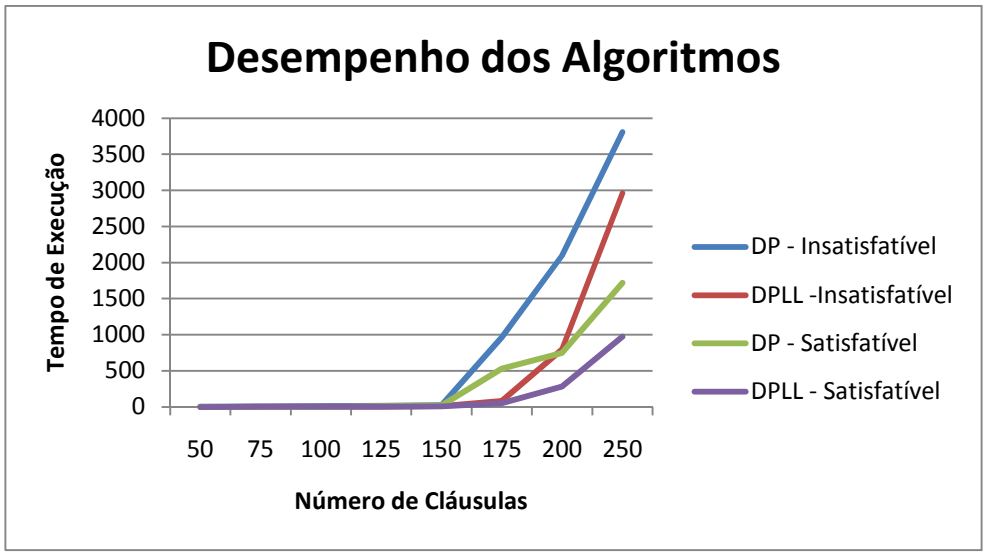

Gráfico 1 - Desempenho Máquina A

O segundo gráfico mostra também o desempenho dos algoritmos, para diferentes instâncias, porém sendo executados pela máquina B.

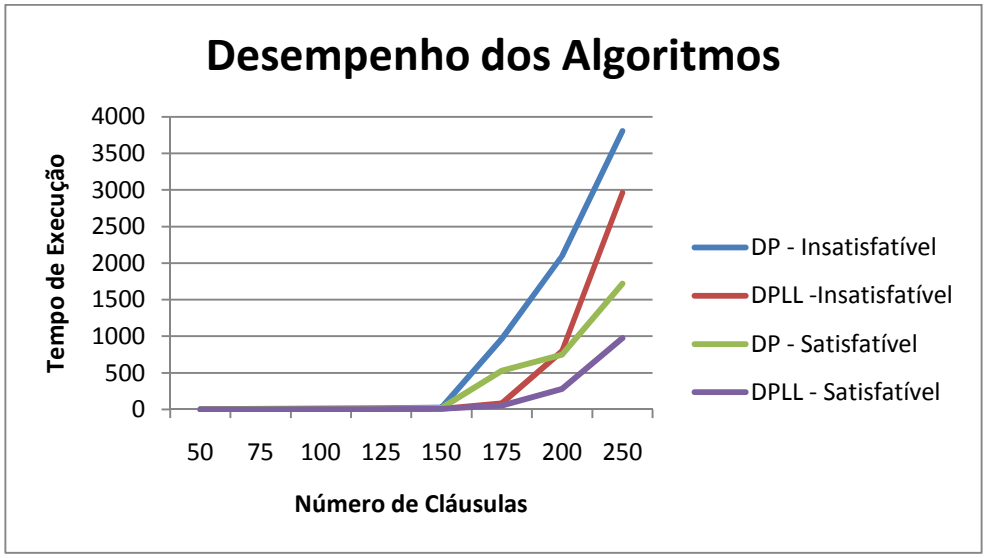

Gráfico 2 - Desempenho Máquina B

Os resultados da execução dos algoritmos se mostraram bastante satisfatórios diante das situações apresentadas, o algoritmo DPLL obteve um desempenho melhor do que o algoritmo Davis-Putnam, pois o primeiro se trata de um aperfeiçoamento do segundo.

Estes algoritmos completos tendem a aumentar exponencialmente o tempo de execução quando é aumentado o número de cláusulas da fórmula.

Podemos perceber que a arquitetura da máquina pouco influenciou no desempenho dos algoritmos, o ganho na velocidade da execução dos algoritmos na máquina A foi um pouco maior do que o desempenho observado da máquina B.

Para as instâncias insatisfatíveis os algoritmos levam mais tempo para verificar a solução, já que, necessitam percorrer e testar todas as variáveis e suas negações presentes na fórmula.

Os algoritmos apresentam bom desempenho para a solução de problemas da satisfatibilidade booleana, mas podem ser melhorados, fazendo com que resolvam em um menor tempo possível, uma fórmula contendo um maior número de cláusulas e variáveis.

5. CONCLUSÃO

O objetivo deste trabalho foi o de investigar as abordagens do problema da satisfatibilidade booleana, os algoritmos que abordam e resolvem esse problema, através de uma análise comparativa entre os algoritmos estudados.

Para isso foi realizado o estudo e implementação dos algoritmos Davis-Putnam e DPLL, utilizando-se a linguagem Python e o *benchmark* SATLIB.

A análise foi feita observando-se o tempo de execução de cada algoritmo através da variação no tamanho da instância de entrada que cada algoritmo executou. Outra análise foi feita observando-se também o tempo de execução que cada algoritmo obteve em diferentes máquinas.

Portanto, foi possível comprovar que dentre os algoritmos estudados, o que melhor soluciona o problema da satisfatibilidade booleana é o algoritmo DPLL, independente da máquina utilizada.

5.1 Trabalhos Futuros

O tema deste trabalho pode ser estudado mais profundamente, através de outras análises ou implementações, assim como:

- Implementar uma variação dos algoritmos clássicos apresentados anteriormente, para melhor resolver o problema SAT.
- Fazer testes com outros algoritmos presentes na literatura, a fim de descobrir o que melhor resolve o problema atualmente.
- Fazer um estudo mais aprofundado na área, principalmente na região do ponto de *crossover*, pois ainda não conseguimos descobrir a dificuldade de um problema sem antes resolvê-lo.

6. REFERÊNCIAS BIBLIOGRÁFICAS

[1] DAVIS, M; PUTNAM, H. **A Computing Procedure for Quantification Theory**. ACM, Julho 1960.

[2] DAVIS, M; LOGEMANN, G; LOVELAND, D. **A machine program for theorem proving**. ACM, Julho 1962.

[3] ZHANG, L; MALIK, S. **The Quest for Efficient Boolean Satisfiability Solvers**. Department of Electrical Engineering, Princeton University, Princeton.

[4] MARQUES-SILVA, J. **Practical Applications of Boolean Satisfiability**. School of Electronics and Computer Science University of Southampton, 2008.

[5] ROQUE, T. **Verificação de Equivalência de Circuitos com Aceleração por Largura e Aprendizado de Cláusulas de Conflito**. Universidade Federal de Minas Gerais, 2007.

[6] WILLEY, J. **Logic for Computer Science: Foundations of Automatic Theorem Proving**. Department of Computer and Information Science, University of Pennsylvania, 2003.

[7] GENESERETH, M. R. AND NILSSON, N. J. **Logical Fundations of Artificial Intelligence**. Morgan Kaufmann Publishers, 1988.

[8] PEREIRA, S. **Apostila Lógica Proposicional**. Universidade de São Paulo.

[9] TONIN, I. **Formas Normais em Lógica de Primeira Ordem. Tese de Doutoramento**. UFSC Universidade Federal de Santa Catarina, 2003.

[10] COOK, S. **The Complexity of Theorem Proving Complexity**. 3th ACM Symposium on Theory of Computing, 151-158.

[11] FERREIRA, F. **Análise Estatística do Problema da Partição Numérica**. Universidade de São Paulo, 2001.

[12] PRESTWICH, S; QUIRKE, C. **Local Search for Very Large SAT Problems**. Department of Computer Science, University College, Cork, Ireland.

[13] ZUIM, R. **Uma heurística de decisão baseada na subtração de cubos para solucionadores DPLL do problema de satisfabilidade**. UFMG Universidade Federal de Minas Gerais, 2007.

[14] CRAWFORD, M.; AUTON, D. **Experimental results on the crossover point in random 3-SAT**. Artificial Intelligence, 1996, volume 81, nº 1-2, págs. 31–57.

[15] GIL, C. **Métodos e técnicas em pesquisa social**. São Paulo: Atlas, 1999.

[16] HOOS, H; STÜTZLE, T. **SATLIB: An Online Resource for Research on SAT**. SAT 2000, pp.283-292, IOS Press, 2000.