Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

PluSAT:
Um resolvedor SAT modular

Autor: Felipe Borges de Souza Chaves
Orientador: Prof. Dr. Bruno César Ribas

Brasília, DF
2022

Felipe Borges de Souza Chaves

PluSAT:
Um resolvedor SAT modular

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Bruno César Ribas

Brasília, DF

2022

Felipe Borges de Souza Chaves
PluSAT:
Um resolvedor SAT modular/ Felipe Borges de Souza Chaves. – Brasília, DF, 2022-
41 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Bruno César Ribas

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2022.

1. SAT. 2. Satisfability. I. Prof. Dr. Bruno César Ribas. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. PluSAT:
Um resolvedor SAT modular

CDU PREVIEW

Felipe Borges de Souza Chaves

PluSAT:
Um resolvedor SAT modular

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 25 de Novembro de 2021 – Data da aprovação do trabalho:

Prof. Dr. Bruno César Ribas
Orientador

Prof. Dr. Edson Alves da Costa Junior
Convidado 1

Prof. Dr. John Lenon Cardoso Gardenghi
Convidado 2

Brasília, DF
2022

Resumo

Este trabalho apresenta um novo resolvedor SAT nomeado PluSAT, um resolvedor com estrutura modular e extensível que possibilita que seus usuários possam modificar o seu comportamento usando *plugins*. A estrutura interna do resolvedor é baseada em resolvedores CDCL (*Conflict-Driven Clause Learning*), nossa interface de extensão possibilita a alteração do comportamento de etapas como: heurística de decisão, retrocessos, BCP (*Boolean Constraint Propagation*), reinícios e análise de conflitos. A estrutura do PluSAT procura oferecer aos desenvolvedores de resolvedores uma forma rápida para implementar esse tipo de solução, enquanto aqueles interessados em iniciar seus estudos podem trabalhar em cada componente separadamente, sem se preocupar com todo o fluxo de um resolvedor comum diminuindo assim a curva de aprendizado necessária para implementar um resolvedor na prática. Toda implementação deste resolvedor está disponibilizada de forma *open source*.

Palavras-chave: satisfatibilidade; SAT; CDCL; DPLL.

Abstract

This work presents a new SAT solver called PluSAT, a solver with a modular structure and pluggable, that makes possible to users change solver behavior without implement all features necessary to create a SAT solver. The internal structure of this solver is based on CDCL (Conflict-Driven Clause Learning), our interface makes possible to change: branch-heuristic, conflict solving, BCP (Boolean Constraint Propagation) and restarts. This architecture seek to provide to developers a fast way to implement kind of algorithm, while newbies can understand what a SAT solver is and can work in each component individually without paying to much attention in all steps that a solver needs to do in the first place. All implementation of PluSAT are open source.

Key-words: satisfability. DPLL. CDCL. SAT.

Lista de ilustrações

Lista de Códigos

Lista de abreviaturas e siglas

BCP	*Boolean Constraint Propagation*
SAT	Satisfatibilidade
CDCL	*Conflict-Driven Clause Learning*
DPLL	Davis–Putnam–Logemann–Loveland
CNF	*Conjutive Normal Form*
TCC	Trabalho de Conclusão de Curso
API	*API Application Programming Interface*
LIFO	*Last In First Out*

Sumário

1 Introdução

O problema de Satisfatibilidade Booleana (SAT) tem uma importância histórica e teórica para a Ciência da Computação. Este problema foi o primeiro a ser provado como NP-Completo pelo teorema de Cook-Levin [1] e por conta dessa característica, cientistas da computação procuram mapear outros problemas para uma instância SAT com o intuito de provar novos problemas NP-Completos.

Satisfatibilidade Booleana consiste em encontrar alguma valoração para uma sentença Booleana que faça com que o resultado da sentença seja verdadeira, uma das técnicas mais difundidas para encontrar possíveis soluções que satisfaçam uma instância SAT são chamadas de resolvedores. Resolvedores SAT modernos realizam buscas no espaço de soluções possíveis, aprendendo durante sua execução cláusulas que podem podar o espaço de busca de possíveis soluções para o problema.

Resolvedores bastante conhecidos como zChaff [2], MiniSAT [3] e Glucose [4] possuem uma estrutura similar na constituição de seus algoritmos, no entanto, cada solução apresenta especificidades que expandem a capacidade desta estrutura comum. A estrutura do PluSAT implementa essa similaridade que resolvedores SAT possuem e disponibiliza componentes para que o usuário possa estender o resolvedor com suas especificações próprias.

1.1 Objetivos

Este trabalho tem o objetivo de apresentar a estrutura do PluSAT, bem como as interfaces que seus usuários podem utilizar para interagir com seus componentes.

O PluSAT procura atender usuários experientes e iniciantes na área de satisfatibilidade. O valor para o público experiente é a facilidade de implementação de testes e a possibilidade de manter várias versões de uma solução com a mesma base, facilitando assim testes e comparações. Para usuários que estão iniciando seus estudos na área, o PluSAT abstrai parte do código necessário para implementar um resolvedor, diminuindo assim a quantidade de código necessário para ter uma versão de resolvedor em funcionamento.

1.2 Estrutura do trabalho

O trabalho é dividido em 5 capítulos sendo este o primeiro, abaixo temos a relação de cada capítulo com seu conteúdo.

- **Capítulo 1:** Introdução do trabalho;
- **Capítulo 2:** Introdução a lógica Booleana, definições e notações básicas;
- **Capítulo 3:** Apresentação das técnicas utilizadas em resolvedores SAT;
- **Capítulo 4:** Estrutura interna e externa do PluSAT;
- **Capítulo 5:** Conclusão.

2 Lógica Booleana

Para definir a o conceito de lógica Booleana primeiro precisamos esclarecer os termos primitivos da mesma.

Axioma 1. Princípio da não contradição: Uma proposição não pode ser verdadeira e falsa ao mesmo tempo.

Axioma 2. Princípio do terceiro excluído: Toda proposição pode ser apenas verdadeira ou falsa, não havendo um terceiro valor possível.

Proposições simples são o componente básico da lógica Booleana, elas são utilizadas para criar proposições compostas. As proposições compostas são várias proposições simples ligadas por um contectivo lógico, os conectivos lógicos da lógica Booleana são o $\wedge$ e $\vee$.

A principal diferença entre ambas é que a proposição simples não possue nenhuma outra proposição em sua composição, enquanto as compostas podem ter duas ou mais proposições.

2.1 Definições e Notações

Instâncias de problemas SAT podem ser definidas como uma proposição composta, neste trabalho será convencionado o conceito de sentença aberta. Uma sentença aberta neste contexto é uma proposição que existe ao menos uma variável não valorada na sentença e em que essas variáveis podem apenas assumir o valor verdadeiro ou falso (seguindo o princípio do primeiro excluído). Nesta definição, a tarefa que o resolvedor procura executar é fechar as sentenças de forma que o resultado da mesma seja verdadeiro.

Variáveis de sentenças são definidas por letras minúsculas do alfabeto como p, q, r e quando numeradas em equações matemáticas serão também definidas por letras minúsculas com um índice, como por exemplo: x_i, p_i e r_i. As variáveis podem se apresentar nas sentenças em sua forma negada ou sem a negação, para diferir esses casos chamamos de literais as diferentes formas que uma variável pode se apresentar. Por exemplo a expressão $(q \wedge \neg q)$ possui apenas uma variável porém dois literais distintos.

Cláusulas de sentenças conjuntivas são definidas por letras do alfabeto maiúsculas, sua indexação seguirá a mesma regra de variáveis porém com letras maiúsculas: C_i, P_i e X_i.

Sentenças e variáveis podem ter valores positivos (denotados por V ou 1) ou valores negados (detonados por F ou 0). Uma sentença só deixa de ser uma sentença aberta (e se torna uma fechada) quando todas suas variáveis tem um valor e assim, a sentença também tem um valor.

2.2 Forma Normal Conjuntiva - CNF

Muitos resolvedores SAT assumem que o formato da sentença segue a Forma Normal Conjuntiva (CNF). Nesta forma, as sentenças são compostas por cláusulas, as cláusulas são definidas como a disjunção dos literais que a compõem como na equação $\vee_{i=1}^{n} v_i$ e as sentenças CNF em si são a conjunção de cláusulas como na equação $\wedge_{j=1}^{m} C_i$.

Com a definição de cláusula podemos definir uma sentença como a conjunção de uma ou mais cláusulas, pela equação $\wedge_{j=1}^{M}(\vee_{i=1}^{N_j} v_{i,j})$ onde a constante M representa o número de cláusulas e a constante N representa o número de literais de cada cláusula.

Apesar da representação lógica atender as necessidades de uma formulação matemática do problema, uma notação baseada em representação de conjuntos é mais prática para definirmos uma estrutura de dados. Nesta representação cada cláusula é composta por um conjunto de literais e a fórmula em si é um conjunto de cláusulas. Podemos exemplificar esse processo convertendo $(p \vee q) \wedge (\neg p \vee q)$ para $\{\{p, q\}, \{\neg p, q\}\}$.

2.3 Considerações Finais

Neste capítulo introduzimos os conceitos básicos de lógica Booleana. Definições e notações que adotaremos pelo resto deste trabalho e apresentamos o formato CNF além de citarmos qualquer expressão lógica pode ser representada por essa fórmula, não transformando a representação em um limitador.

3 Satisfatibilidade

Nesta seção iremos definir os principais algoritmos e teorias utilizadas na composição de um resolvedor SAT, os algoritmos apresentados nesta seção tem o foco didático e em um resolvedor SAT competitivo, esses algoritmos são implementados com técnicas que prezam pela eficiência, podendo diferir bastante do pseudocódigo apresentado.

Existem diversos tipos de resolvedores SAT, neste trabalho iremos nos concentrar em resolvedores completos (resolvedores que sempre encontram um resultado para a sentença, sendo entre os possíveis satisfatível ou insatisfatível). Na classe de resolvedores completos iremos focar em resolvedores que utilizam como base o algoritmo de DPLL e BCP como Minisat [3] e Glucose [4].

Por fim, iremos dividir o resolvedor em três componentes principais: Representação da sentença, Inferência e Busca.

3.1 Representação de Sentença

Existem representações alternativas à CNF como a utilizada pelo LIAMFSAT [5], que não necessita de transformações na sentença original. Porém, grande parte dos resolvedores CDCL que citamos até aqui como o Minisat e o zChaff consomem exclusivamente sentenças em arquivo no formato DIMACS CNF [6].

No cabeçalho do arquivo temos a definição do problema no formato *p cnf VAR CLS*. O prefixo desta linha indica que isto é uma definição de problema CNF (o formato DIMACS tem suporte para outros tipos de arquivos, porém utilizamos apenas CNF neste trabalho). VAR é uma constante que indica o número de variáveis presentes em sua sentença, enquanto CLS determina o número de cláusulas que a mesma irá conter.

A definição de cada cláusula deve ser feita após a definição do problema. Para representar uma cláusula é preciso listar todos os literais que a cláusula contém. Primeiramente todas as variáveis na sentença recebem um índice entre os valores $[1..N]$, na definição da cláusula, cada literal é indicado por esse índice e no caso dos literais negados fazemos com que o índice tenha o sinal negativo. Para finalizar a definição da cláusula basta adicionar o terminador 0(zero) ao final da definição.

Para exemplificar melhor vamos transformar a Equação 3.1 em um arquivo CNF, representado no Código 1.

$$(a \vee b) \wedge (a \vee d) \wedge (\neg a \vee \neg b \vee c \vee d) \tag{3.1}$$

```
c this is a comment
p cnf 4 3
1 2 0
1 4 0
c split clause in multiple lines
-1 -2
3 4 0
```

Código 1: Exemplo de arquivo CNF

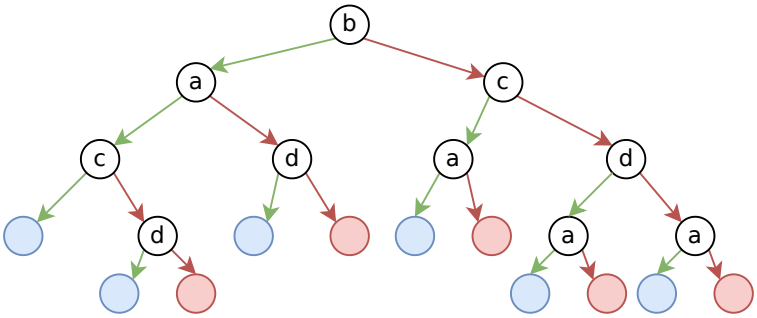

Figura 1 – Árvore de busca da equação 3.1

Adicionalmente como exemplificado no Código 1 podemos ter comentários no documento, qualquer linha que se inicie com o caractere *c* são ignoradas durante o processo de leitura.

3.2 Busca

Para sistematicamente percorrer o espaço de interpretações de uma sentença CNF, os resolvedores SAT completos utilizam o algoritmo DPLL [7]. Em cada interação esse algoritmo escolhe uma variável e decide um valor para mesma, continuando esse processo até que todas tenham um valor (feche a sentença). Caso essas decisões levem a uma contradição lógica ou a uma valoração falsa, a busca desfaz as decisões e realiza outras decisões com novos valores.

Em termos práticos, a decisão de valores para as variáveis da sentença pode ser interpretada como uma busca por profundidade em uma árvore. Essa árvore é chamada de árvore de busca [8]. Na Figura 1 temos uma árvore de busca para a Equação 3.1 onde cada nó é uma variável e setas vermelhas correspondem a valorações falsas e setas verdes valorações verdadeiras da mesma.

Cada nível desta árvore é chamado de nível de decisão, durante o processo de busca o resolvedor mantém todas as decisões na ordem inversa que foram realizadas, ao encontrar um conflito o resolvedor desfaz decisões para que possa testar outros valores.

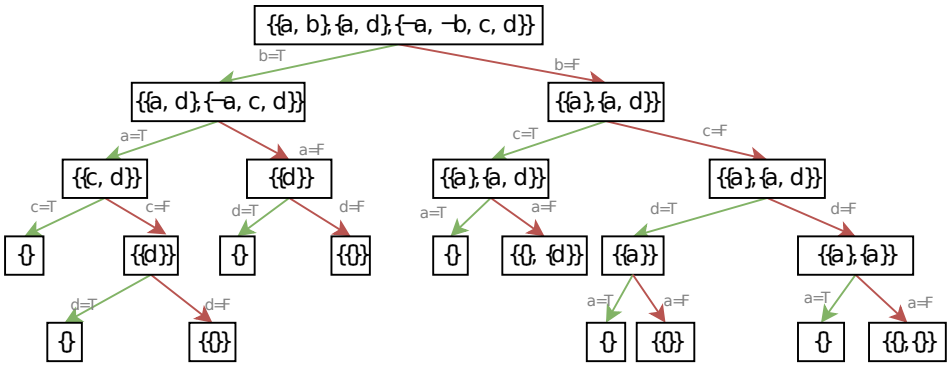

Figura 2 – Execução do algoritmo DPLL com condicionamento

Nas próximas seções será apresentada a principal configuração que um resolvedor assume para busca em grande parte dos resolvedores. Essa configuração consiste em definir valores para variáveis via DPLL, calcular as consequências de cada valoração via um mecanismo chamado de BCP e caso a valoração leve alguma cláusula da sentença para condição de conflito, os valores definidos são revertidos e outros são testados.

3.2.1 Condicionamento ou Boolean Constraint Propagation

A valoração de uma variável é definida pela operação de condicionamento. Condicionamento consiste em remover cláusulas ou literais da sentença a partir do valor de sua variável, gerando assim uma sentença derivada [8] como definida na equação 3.2 onde Δ é a sentença, L é o literal sendo valorado e α é uma cláusula da sentença.

$$\Delta|L = \{\alpha - \{\neg L\} | \alpha \in \Delta, L \notin \alpha\} \tag{3.2}$$

O processo de condicionamento se inicia quando escolhemos uma valoração para um literal. Após isso propagamos o resultado seguindo as seguintes regras:

- Caso a valoração seja o valor falso, removemos esse literal de todas as cláusulas do conjunto
- Caso a valoração seja positiva, removemos as cláusulas que contém esse literal da sentença.

O mecanismo de condicionamento gera uma nova sentença com base nos valores que seus literais assumiram. Esse mecanismo será utilizado na etapa de valoração por DPLL para calcular as consequências de sua valoração, para exemplificar, a execução do DPLL na equação 3.1 pode ser representada na Figura 2.

3.2.2 DPLL

O algoritmo Davis-Putnam-Logemann-Loveland (DPLL) [7] combina a busca por valores de variáveis e o condicionamento de sentenças para sistematicamente percorrer o espaço de possíveis soluções de uma instância SAT.

O algoritmo DPLL contém 3 regras principais [7]:

1. **Remoção de cláusulas unitárias**: Cláusulas com apenas uma variável em uma sentença precisam necessariamente receber o valor verdadeiro para que a sentença seja satisfeita. Essa regra identifica essas cláusulas e propaga os valores em toda a sentença.
2. **Literal puro**: Caso uma variável possua apenas literais negados ou apenas literais positivos, podemos retirar todas as cláusulas que possui essa determinada variável, pois podemos definir um valor para essa variável de forma que não fique conflitante com o restante e que faça a cláusula positiva.
3. **Ramificação**: A regra de ramificação busca avaliar ambos os valores possíveis para uma variável a fim de verificar a consequência dessa decisão.

A primeira regra é implementada pela inferência (dedicaremos a próxima seção a esta). Na prática a segunda regra pode ser executada antes da execução do algoritmo do DPLL como um pré-processamento diretamente na sentença CNF. Resta exemplificar a terceira e última regra que efetivamente é a regra que irá gerar a árvore de busca, no Código 2 se encontra um pseudocódigo de uma implementação da regra de ramificação.

Inicialmente o algoritmo verifica os casos base que uma sentença pode chegar via condicionamento. Caso a sentença esteja vazia isso significa que o mecanismo de condicionamento conseguiu retirar todas as cláusulas e portanto a sentença foi satisfeita. Caso a sentença possua o apenas literais falsos, isso significa que alguma cláusula teve todos os seus literais valorados como falso e portanto essa valoração é insatisfatível.

O algoritmo continua escolhendo uma variável para ser utilizada na próxima interação por meio de heurísticas de decisão em seguida, a satisfatibilidade é verificada novamente, este processo continua até que todas as alternativas seja esgotadas.

Este algoritmo consegue percorrer todos os nós da árvore de busca, porém, podemos adicionar outros mecanismos que podem podar ramos dessa árvore, como o já citado mecanismo de inferências e outros que iremos apresentar no contexto de busca.

3.2.3 Heurísticas de ramificação

Para eleger uma variável para valoração do DPLL é utilizado uma heurística. As heurísticas podem assumir diversas estratégias como algoritmos aleatórios [8] e até

```
1  bool DPLL(Sentenca s)
2
3      se: sentença s não tiver mais cláusulas
4          retorno Verdadeiro
5
6      se: alguma cláusula em s tiver todos literais falsos
7          retorno Falso
8
9      i <- EscolhaVariavel()
10
11     nNovaS = Condicionamento(Literal(i), s)
12     negNovaS = Condicionamento(LiteralNegado(i), s)
13
14     se: DPLL(nNovaS) ou DPLL(negNovaS) for Verdadeiro
15         return Verdadeiro.
16
17     return Falso
```

Código 2: Pseudocódigo DPLL recursivo

estratégias baseadas na frequência que literais são avaliados [2, 5].

A importância de uma boa heurística de ramificação está em situações onde a ordem das variáveis podem reduzir o tamanho dos caminhos da árvore de busca [8]. Por exemplo na Figura 1, após a valoração da variável b o algoritmo segue fazendo a valoração da variável c que neste momento não está em nenhuma cláusula ainda removida. Podemos continuar esse exemplo avaliando que, após realizar a valoração de b a melhor opção seria valorar a variável a e isto deixaria essa ramificação com apenas dois níveis de decisão.

3.2.4 Retrocesso

Quando o algoritmo do DPLL encontra um valoração com a qual ele não pode prosseguir, o retrocesso é realizado para desfazer uma ou mais decisões, para que dessa forma o DPLL possa seguir.

Existem dois tipos de retrocessos: cronológico e não cronológico. O cronológico desfaz a última decisão feita pelo algoritmo do DPLL. O retrocesso não cronológico pode retroceder para qualquer nível anterior, sem a limitação de ser o último. O retrocesso não cronológico é útil caso você tenha garantia de que qualquer combinação de valores a partir daquela decisão vai resultar em uma solução insatisfativel.

3.3 Inferência

Inferência em resolvedores são mecanismos que tentam assumir parte do trabalho do DPLL em situações onde não existiria uma alternativa possível ou o trabalho feito pela busca vai levar apenas a resultados insatisfatíveis.

Nesta seção será apresentado alguns destes mecanismos. O primeiro é a propagação unitária, que assina valores a variáveis em situações onde um valor alternativo levaria a uma resposta insatifatível. Outro mecanismo de inferência é o aprendizado de cláusulas, combinado com a propagação unitária o resolvedor pode modificar a sentença, adicionando novas cláusula que são cuidadosamente montadas para evitar caminhos que já foram explorados e já existe a certeza de que terá um resultado insatisfatível.

3.3.1 Propagação unitária

Como já definimos, para que a sentença seja satisfatível, todas as cláusulas precisam necessariamente ter ao menos um literal verdadeiro. Neste cenário, se temos uma cláusula com um literal ainda não valorado e todos os restantes com o valor falso, podemos assumir que o valor para esse último literal necessariamente deve ser verdadeiro. Para essa condição, onde uma cláusula tem um literal não valorado e todos os restantes falsos dizemos que essa cláusula está em estado crítico.

Os autores do resolvedor zChaff [2] perceberam que em cláusulas críticas podemos pular passos e decidir o valor verdadeiro para a variável ainda não valorada na cláusula. Dessa forma, não é necessário avaliar outro valor para esse literal.

Esse fato faz com que o resolvedor não precise gastar processamento em alguns caminhos de decisão que vão levar a interpretações falsas da fórmula CNF, diminuindo o número de valorações feita pela busca do resolvedor.

Na Figura 3 ilustramos a execução do Código 2 para a sentença representada na Equação 3.1.

3.3.2 Aprendizado em resolvedores

A ideia de aprendizado em resolvedores SAT se baseia no fato que durante a execução do resolvedor podemos identificar algumas propriedades da fórmula CNF e reaproveitar estas propriedades para podar caminhos na árvore de valoração.

O primeiro passo para atingir aprendizado é identificar a causa dos conflitos durante a valoração [8]. Para isso mantemos uma estrutura chamada de grafo de implicações que relaciona a valoração de uma variável com as consequentes propagações unitárias. Com isso conseguimos reconhecer quando determinada valoração de um conjunto de literais faz com que a fórmula fique negativa, dizemos que essas valorações são conflitantes entre si.

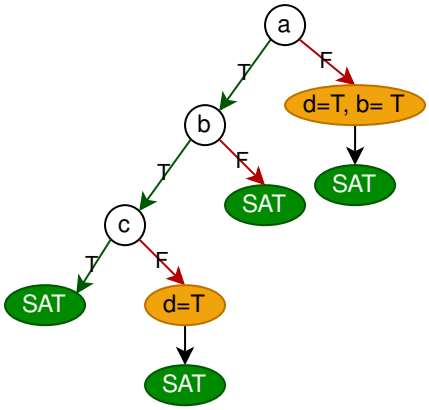

Figura 3 – Exemplo de execução do DPLL

Uma vez identificada as variáveis e valores que são conflitantes, precisamos seguir para a etapa de aprendizado, onde adicionamos cláusulas dentro da fórmula CNF para que o resolvedor reconheça que aquele é um caminho conflitante.

3.4 Considerações finais

Neste capítulo foram apresentados os blocos construtores de um resolvedor e uma breve noção de como eles interagem juntos. É importante notar que alguns desses passos não são necessários em qualquer resolvedor. É perfeitamente factível construir um resolvedor sem mecanismos de inferência, com uma heurística de ramificação simples e sem nenhum aprendizado de cláusulas, porém, apesar de não serem obrigatórios cada um destes componentes colaboram para uma execução mais eficiente do resolvedor.

Como ilustrado, mecanismos de inferências tem uma grande importância para podar caminhos na árvore de valoração na busca. Além de mecanismos de inferência, um bom algoritmo de ramificação pode fazer com que o DPLL encontre o resultado com menos decisões e propagações do BCP e como evidenciado, o BCP é o processo mais custoso em um resolvedor e portanto técnicas que diminuem o seu uso trazem uma melhoria grande na performace do resolvedor.

4 PluSAT

O PluSAT é um resolvedor modular, sua arquitetura é dividida em duas partes principais: arquitetura interna e arquitetura externa.

A primeira é a arquitetura interna, que contém a lógica e estrutura de dados para executar tarefas como o DPLL, leitura da sentença e retrocesso. O desenvolvedor terá acesso a interface interna, porém, não será o responsável por executar ou controlar a lógica do mesmo. Quem executa a estrutura interna é o próprio resolvedor, enquanto o usuário pode interagir com a mesma.

A segunda é a arquitetura externa, aqui contém o código usuário. O PluSAT não tem controle do código do usuário, a missão do resolvedor é apenas chamar a implementação do usuário quando necessário. Durante o desenvolvimento de seu código o usuário irá interagir com o resolvedor a partir de uma Application Programming Interface (API) com o PluSAT.

Por fim, a última definição que precisamos discorrer sobre é como o usuário irá adicionar seu código junto ao código do resolvedor e que cenários são previstos nesta estrutura.

4.1 API

Esta seção se dedica a definir os tipos e as interfaces do PluSAT [1]. Cada tipo possui uma coleção de funcionalidades que manipulam determinada estrutura de dados. Todas as funções da API do PluSAT estão listadas no Apêndice A.

4.1.1 Cláusulas

A interface de cláusulas é responsável por organizar cláusulas em listas e realizar a criação e limpeza destes dados na memória. No Código 3 temos a especificação desta interface, no código do PluSAT é possível encontrar esse conteúdo no cabeçalho cujo nome do arquivo é *formula.h*.

4.1.2 Sentença

A interface de sentenças definida no Código 4 dispõe de funções para adicionar cláusulas à sentença, buscar as cláusulas ligadas a um literal e limpar a memória da sentença dentro do resolvedor.

[1] PluSAT em https://github.com/UnB-SAT/PluSAT

```c
/* Literais podem apenas ser assinados como FALSE, TRUE e UNK */
typedef enum LiteralStates{FALSE, TRUE, UNK} LitState;

/* Literais e Variáveis tem 16 bits sinalizado */
typedef int16_t LiteralId;
typedef int16_t VariableId;

/* Estrutura de uma cláusula, o tamanho do vetor
 * de literais é definido a partir do número de literais
 * da cláusula, após a criação edições não são possíveis via interface */
typedef struct Clause{
    uint8_t size;
    LiteralId* literals;
}Clause;

/* A lista de cláusulas é uma lista ligada simples,
 * cada nó contém uma referência para uma cláusula */
typedef struct Node{
    Clause* clause;
    struct Node* next;
}ClauseNode;

/* Auxilia na criação de uma nova cláusula */
Clause* newClause(LiteralId*, uint8_t);

/* Limpa a estrutura da memória */
void freeClause(Clause*);

/* Adiciona um elemento a lista de cláusulas
 * em tempo constante, o novo nó da lista se torna o primeiro */
ClauseNode* addNodeOnList(Clause*, ClauseNode*);

/* Limpa a lista e as cláusulas referenciadas pela mesma da memória */
void freeList(ClauseNode*);
```

Código 3: API para interação com cláusulas

```c
/* Estrutura da formula, a tabela de
 * literais indexa literais não negado em índices pares
 * e literais negado em índices ímpares
*/
typedef struct Form{
    int16_t numClauses;
    int16_t numVars;

    ClauseNode* clauses;
    ClauseNode** literals;
}Form;

/* Cria sentença vazia com o número de variáveis enviadas no argumento */
Form* newForm(uint16_t);

/* Limpa variável da memória */
void freeForm(Form*);

/* Adiciona uma cláusula na estrutura da sentença */
void addClause(Clause*, Form*);

/* Retorna a lista de cláusulas ligadas a um determinado literal na tabela de literais */
ClauseNode* getLiteralClauses(const LiteralId lit, const Form* form);
```

Código 4: API para interação com a sentença

4.1.3 Decisão

O último aspecto da API do PluSAT é o controle sobre as decisões do DPLL. No Código 6 apresentamos as interfaces disponíveis para interagir com a estrutura interna de decisões do resolvedor. Dentro do Código 5 temos a representação da estrutura interna de decisões, após a realização da leitura do arquivo o PluSAT inicia essas estruturas com base nas informações definidas no arquivo como o número de variáveis.

4.2 Estrutura

Para construir um resolvedor SAT, é necessário entender como todas as técnicas apresentadas no capítulo anterior interagem entre si para buscar uma solução.

Inicialmente temos a representação da fórmula CNF que se encarrega de estruturar os dados de entrada do resolvedor. A partir deste ponto o algoritmo pode realizar qualquer pré-processamento e iniciar o algoritmo de DPLL. Cada iteração do algoritmo utiliza a heurística de ramificação para escolher a variável a ser valorada naquele momento, restando ao DPLL escolher um valor para essa variável e iniciar o BCP, que por sua vez propaga a decisão para os literais na fórmula.

```c
/* Possíveis resultados do algoritmo DPLL */
enum SolverResult {UNSAT, SAT};

/* Estados que o módulo de decisão pode notificar para o
 * resolvedor, o estado ALL_ASSIGNED faz com que caso
 * não existe nenhum conflito a sentença seja considerada SAT */
enum DecideState {ALL_TRIED, FOUND_VAR, ALL_ASSIGNED};

/* Registro de uma decisão do algoritmo DPLL*/
typedef struct Decision
{
    VariableId id;
    uint16_t value;
    bool flipped;
    struct DecisionList *consequences;
}Decision;

/* A consequência é uma lista de decisões */
typedef struct DecisionList
{
    Decision *decision;
    struct DecisionList *next;
}DecisionList;
```

Código 5: Estruturas de dados da interface de decisão PluSAT

Até este ponto temos o fluxo direto do resolvedor, no entanto, ao encontrar condições insatisfatíveis o mecanismo de BCP identifica o conflito, o conflito é analisado e caso exista um esquema de aprendizado de cláusulas ele é requisitado para adicionar as cláusulas necessárias. Após finalizar o aprendizado o resolvedor irá retroceder as decisões. Esse processo continua até que o resolvedor tente retroceder no primeiro nível da árvore de busca (sendo assim a sentença UNSAT) ou que o DPLL encontre uma solução para a sentença (solução SAT).

Em um resolvedor modular, a lógica do algoritmo se mantém, porém, é necessário desagregar as interfaces extensíveis do código principal. O PluSAT separa os seus componentes como ilustrado na Figura 4, os componentes em cinza são componentes da estrutura externa e os pretos são da estrutura interna. Perceba que o PluSAT é um resolvedor completo, portanto os dois únicos resultados que ele pode chegar é UNSAT ou SAT.

```
1  /* Retornar uma inteiro representando o número de decisões atuais */
2  int getLevel();
3
4  /* Empilha uma decisão da estrutura interna do resolvedor */
5  void insertDecisionLevel(const VariableId, const int);
6
7  /* Retorna para o nível de decisão enviado no argumento da função (retrocesso)*/
8  void backtrackTo(uint16_t);
9
10 /* Retorna a última decisão feita, essa função não altera a estrutura de decisão */
11 Decision* getLastDecision();
12
13 /* Retorna o estado de um determinado literal */
14 enum LiteralStates getLitState(LiteralId);
15
16 /* Valora uma variável sem criar uma decisão */
17 void setVarState(VariableId, LitState);
18
19 /* Retorna o estado atual de uma variável */
20 LitState getVarState(const VariableId var);
21
22 /* Permite que o usuário acesse a pilha de decisão diretamente,
23  * este é um vetor estático com o tamanho máximo sendo o número de
24  * variáveis da fórmula */
25 Decision* getDecisions();
```

Código 6: API para interagir com estrutura de decisão interna

4.3 Estrutura Interna

A estrutura interna é a base da interface do PluSAT, o usuário poderá interagir com ela via API porém não poderá alterá-la. A seguir será é apresentado a proposta desta estrutura.

4.3.1 Representação

Inicialmente as fórmulas CNF estão representadas em arquivos de extensão cnf. Esses arquivos são utilizados como entrada para o resolvedor. A leitura do arquivo ignora todas as linhas de comentário e tem suporte para separação de cláusulas em múltiplas linhas do arquivo. A definição do número de variáveis e cláusulas deve ser apresentadas antes de qualquer cláusula.

A leitura do arquivo CNF vai estruturar a representação do arquivo em uma estrutura do resolvedor. Essa estrutura possui duas outras estruturas internas: tabela de literais e lista de cláusulas.

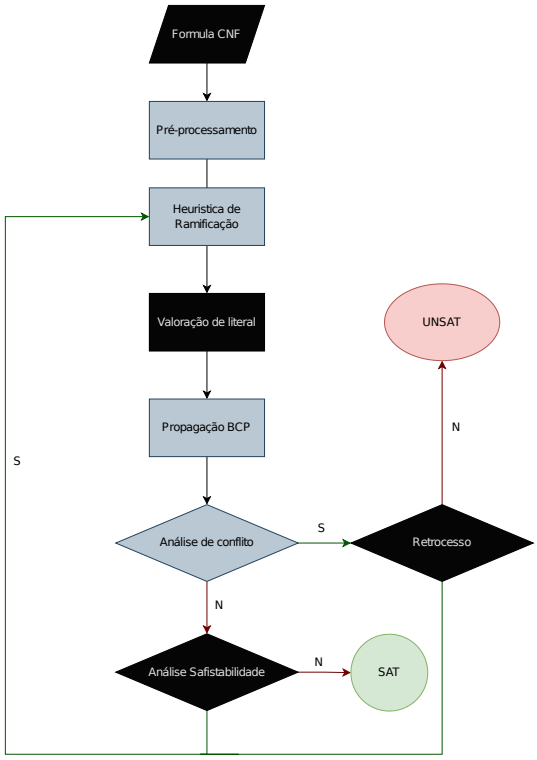

Figura 4 – Arquitetura do PluSAT. Componentização interna e externa

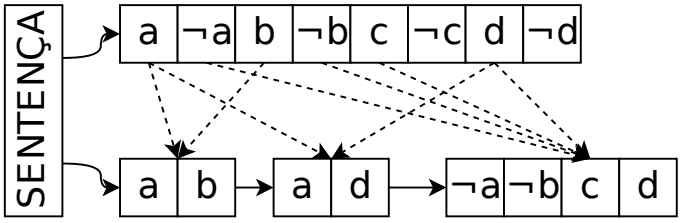

Figura 5 – Representação da sentença CNF pelo PluSAT

A tabela de literais é uma tabela que indexa os literais negados nos índices ímpares e os não negados em índices pares. A motivação dessa estrutura é para rapidamente encontrar todas as cláusulas que contém um determinado literal, como foi pontuado na seção anterior, o valor do literal irá determinar se a cláusula continuará na fórmula ou poderá ser removido (ou ignorado).

A lista de cláusulas é apenas uma lista simples que agrega todas as cláusulas juntas. Essa lista é uma forma rápida de manter todos as cláusulas em memória e facilmente removê-los ao final do programa. Além disso, ter apenas uma instância da cláusula na memória e referenciar os literais pela tabela de literais é uma forma mais eficiente de utilizar a memória do resolvedor do que replicar todas as cláusulas para todos os literais.

A estrutura interna está representada na Figura 5.

Para interagir com este módulo interno o usuário dispõe de algumas interfaces. A estrutura da cláusula é composta por um vetor estático de literais, onde cada literal é

```
1 // LiteralId é um outro nome para uint16_t
2 LiteralId values[] = {1, 2, 3};
3
4 // Vetor de literais e a quantidade de literias do mesmo
5 Clause* clause = newClause(values, 3);
6
7 // Libera a memória (descessário caso seja adicionado à estrutura Form)
8 freeClause(clause);
```

Código 7: Exemplo de criação de cláusulas no PluSAT.

representado por um inteiro de 16 bits sinalizado, desta forma a cláusula não foi desenhada para ser editada.

O processo de se criar e deletar uma cláusula pode ser visualizado no Código 7. Perceba que a cláusula não necessariamente está ligada à representação da sentença, esse processo foi intencionalmente desenhado para que desenvolvedores de aprendizado de cláusulas possam ter as referências das suas próprias cláusulas em tempo de execução, isso facilita a diferenciação de cláusulas criadas pelo PluSAT automaticamente e cláusulas criadas pelo usuário. Caso o usuário adicione a cláusula à fórmula CNF, não é preciso se preocupar com a limpeza da memória pois o PluSAT ficará responsável por esse trabalho.

Com a cláusula criada é possível interagir com a fórmula, a operação de adicionar uma cláusula nesta estrutura é bem simples, inicialmente a cláusula é adicionado a lista em tempo constante (cada novo nó na lista se torna o início da lista), após esse processo o resolvedor percorre todos os literais adicionando uma referência desta cláusula na posição do literal na lista de literais.

Apesar de simples, é um processo bem sensível a erros, qualquer índice errado ou problema na hora de referenciar a nova cláusula da lista de cláusulas pode invalidar todo o processo do resolvedor. Visto a importância deste processo também é disponibilizado na API do PluSAT uma forma de criar uma nova fórmula e adicionar cláusulas a ela, fazendo todo esse processo descrito anteriormente de forma transparente para o usuário. No Código 8 existe um exemplo do uso desta interface.

Como evidenciado no exemplo, para iniciar uma sentença é preciso definir o número de variáveis que essa sentença vai ter, a necessidade de definir esses valores estaticamente se dá pela tabela de literais, a mesma é uma tabela estática com duas vezes o tamanho do número de variável (pois é possível apenas ter dois literais para cada variável), lidar com uma tabela dinâmica seria complexo e não traria tanto valor para o desenvolvedor, pois a maioria das técnicas buscam diminuir o número de variáveis de uma sentença.

Por último, para encontrar a posição de um literal específico dentro da tabela de literais, o PluSAT dispõe da função *getLiteralClauses*, que retorna um ponteiro para

```
LiteralId values[] = {1, 2, 3};

// Nova sentença sem nenhuma cláusula com 3 variáveis
Form* form = newForm(3);

// Cria uma nova cláusula
Clause* clause = newClause(values, 3);

// Adiciona uma nova cláusula à fórmula
addClause(clause, form);

// Limpa a memória das cláusulas e a tabela de literais
freeForm(form);
```

Código 8: Exemplo de fórmula com cláusulas

lista de literais e recebe um LiteralId.

4.3.2 DPLL

O DPLL em resolvedores implementa uma busca por profundidade, essa busca utiliza de módulos internos como retrocesso e verificação de satisfatibilidade. Além disso, o resolvedor faz chamadas aos módulos escritos pelo usuário, como os módulos de heurísticas de decisão, BCP e resolução de conflitos.

A lógica deste componente está expressa no Código 9. Para realizar a busca o primeiro passo é decidir uma variável, para isto é utilizado o módulo de heurística de ramificação escrito pelo usuário, representado aqui na função *Decide*. Este módulo escolhe a variável e retorna o estado para o resolvedor, revelando se todas as variáveis foram decididas ou se uma nova foi encontrada. Após isso o algoritmo continua utilizando o módulo BCP (também definido pelo usuário) caso o mesmo encontre algum conflito é o mecanismo de resolução de conflito é acionado, representado aqui pela função $ResolverConflito$, caso o módulo consiga resolver o conflito ele precisa indicar quantos níveis é preciso retornar para que o conflito seja resolvido, caso não seja possível o número de níveis a retornar é sempre 0. Por fim, caso todas as variáveis tenha sido valoradas o resolvedor considera a sentença como SAT, caso contrário, o processo continua.

4.3.3 Retrocesso

Para realizar o retrocesso o PluSAT mantém uma estrutura *Last In First Out*($LIFO$) para registrar a sequência de decisões feitas pelo resolvedor, cada elemento na pilha representa um nível de decisão do resolvedor. Quando utilizamos propagação unitária podemos acabar tendo decisões que são consequências de uma decisão feita pelo resolvedor, para

```
enum ResultadoDPLL DPLL(Sentenca *problem){
    enum EstadoDecisao dState;

    while(true){
        dState = Decide(problem);

        while(BCP(problem) == TEM_CONFLITO){
            voltarNNiveis = ResolverConflito();

            if(voltarNNiveis == 0)
                return UNSAT;
            else
                retrocederPara(voltarNNiveis)
        }

        if(dState == ALL_ASSIGNED)
            return SAT;
    }
}
```

Código 9: Implementação da lógica do DPLL

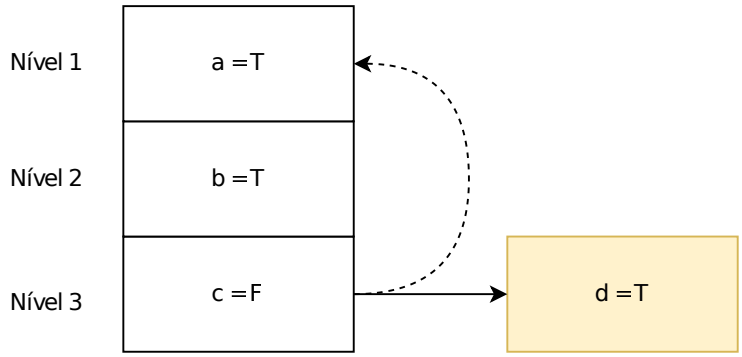

Figura 6 – Pilha de decisão de um resolvedor, a linha pontilhada indica retrocesso

facilmente remover a decisão e suas consequências, elas são adicionadas no mesmo nível que a decisão principal.

Ilustramos a pilha de decisão para uma das possíveis soluções da Figura 3 na Figura 6, perceba que a variável d é valorada como consequência da decisão de valorar c como verdadeiro, portanto, d é uma consequência que será registrada no mesmo nível de decisão que c.

Nesta estrutura, o processo de retrocesso se reduz a simplesmente limpar os elementos da fila até encontrar o nível desejado, ainda na Figura 6, ilustramos com a seta pontilhada o processo de retrocesso não cronológico. Neste retrocesso, estaríamos retornando dois níveis e tiraríamos a valoração das variáveis c, d (ambas no nível 3) e b (nível

```
1 VariableId var = 1
2
3 // Valora variável e insere a mesma da pilha de decisão
4 insertDecisionLevel(var, FALSE)
5
6 // Retorna para o nível 1
7 backtrackTo(1)
```

Código 10: Interface para adição e removeção de decisões

2).

No Código 10 está exemplificado como inserir decisões na pilha por meio da interface *insertDecisionLevel* e como fazer o retrocesso por meio do método *backtrackTo*.

4.3.4 Verificação de Satisfatibilidade

Este módulo interno é bem simples mas com uma semântica bastante importante. Quando definimos o algoritmo de DPLL e BCP até aqui, demonstramos que a satisfatibilidade de uma sentença está estritamente ligada às condições das cláusulas durante a execução do resolvedor. Essa abordagem é correta, porém pouco prática, para verificar a condição das cláusulas o resolvedor precisa manter estruturas para rapidamente identificar se cada cláusula foi satisfeita e essa condição pode mudar a cada retrocesso ou decisão do resolvedor.

Uma forma mais prática é definir se a solução foi satisfeita ou não com base no número de variáveis valoradas com sucesso. Se conseguimos valorar todas as variáveis da nossa sentença sem encontrarmos conflitos, isso indica que a sentença é SAT. A vantagem de interpretar a satisfatibilidade dessa forma é que precisamos preocupar com menos pontos, basta verificar a cada decisão se existe uma nova variável não valorada e caso essa decisão não seja correta, o retrocesso cuidará de voltar o resolvedor para um estado que ele possa prosseguir com a busca

4.4 Estrutura Externa

A estrutura externa dá suporte para o usuário criar componentes do resolvedor e injetar este código na execução do PluSAT.

Como esses módulos são dedicados a implementação do usuário, seria muito difícil listar todas as possibilidades de implementação e uso de cada um. Nesta seção vamos nos concentrar em declarar a funcionalidade do módulo e demonstrar alguns exemplos e cenários que foram considerados na concepção do mesmo.

```
1  void PreProcessing(Form* form){
2
3      for(int i = 0; i < 2*form->numVars; ++i){
4          if(form->literals[i] == NULL){
5              if(i%2==0 && form->literals[i+1] != NULL)
6                  setVarState((int)i/2, TRUE);
7
8              if(i%2!=0 && form->literals[i-1] != NULL)
9                  setVarState((int)((i+1)/2), FALSE);
10         }
11     }
12 }
```

Código 11: Exemplo de implementação interface de Pré processamento

4.4.1 Pré processamento

O módulo de pré processamento é acionado antes do início da execução do DPLL. Esse módulo tem como entrada a sentença CNF, como no Código 11.

No Código 11, temos um exemplo de pré processamento onde é verificado se algum literal não tem sua contraparte negada. Caso um literal aparece somente na mesma forma, basta valorarmos o mesmo como verdadeiro que temos a garantia de que essa decisão não causaria nenhum conflito. Ainda no exemplo, recebemos a sentença na chamada da função, verificamos a tabela de literais e decidimos o valor da variável utilizando a interface $setVarState$, a chamada desta função não popula a pilha de decisão do resolvedor, isso significa que esta decisão não vai ser retrocedida durante a execução do PluSAT. Em resumo, estamos fazendo uma operação de condicionamento no pré processamento, neste caso, a sentença processada pelo PluSAT não é a mesma enviada no arquivo.

O usuário pode utilizar esse módulo por vários motivos: remover cláusulas unitárias, identificar variáveis que possuem apenas um mesmo literal em toda a sentença e manipular a tabela de literais para mecanismos de observação. Algumas configurações de resolvedores não precisam utilizar este módulo, portanto é possível apenas gerar uma implementação vazia que não modifique a fórmula.

4.4.2 Heurísticas de Ramificação

O módulo de heurística de ramificação tem acesso a representação da fórmula CNF, a missão deste módulo é inserir a decisão na estrutura interna do resolvedor e notificar o algoritmo do DPLL sobre o estado da decisão, que se resume a notificar se uma variável foi encontrada ou se todas foram valoradas.

No contexto de verificação de satisfatibilidade, foi definido que o PluSAT verifica

```
1  enum DecideState Decide(const Form* form){
2      ClauseNode* list = form->clauses;
3
4      while(list != NULL){
5          Clause *clause = list->clause;
6
7          for(int i = 0; i<clause->size; ++i){
8              LiteralId lit = clause->literals[i];
9              lit = ((lit > 0 )? lit : -lit);
10
11             if(getVarState(lit-1) == UNK){
12                 insertDecisionLevel(lit-1, FALSE);
13                 return FOUND_VAR;
14             }
15         }
16         list = list->next;
17     }
18     return ALL_ASSIGNED;
19 }
```

Código 12: Exemplo de implementação de heurística de decisão

se todas as variáveis foram valoradas para determinar se a solução é SAT. Neste cenário, a implementação do usuário deve verificar se todas as variáveis foram valoradas, com base nisso, o PluSAT verifica se o mecanismo de BCP não encontrou algum conflito, caso não encontre algum conflito ou o conflito seja resolvido pelo módulo de resolução de conflitos o resolvedor assume a sentença como SAT, como definido no Código 9.

No Código 12 temos um exemplo de implementação da interface de Heurística de Ramificação, o usuário implementa a lógica da ramificação pela interface *Decide*. Essa implementação insere a valoração do literal e retorna para o resolvedor o estado da decisão, que indica se uma variável foi encontrada ou se todas já estão valoradas. Nesta implementação de exemplo, a heurística seguida é assinar as variáveis na ordem que aparecem nas cláusulas da sentença.

No Código 12 é feita uma decisão de valoração de uma variável e essa decisão é registrada na pilha de decisão do resolvedor por meio da API *insertDecisionLevel*, perceba que o primeiro argumento representa o valor do literal positivo subtraído por um, essa substração é feita pois no PluSAT as variáveis são indexadas a partir de 0, enquanto literais são indexados a partir de 1 para ser compatível com com a representação da sentença.

4.4.3 BCP

O módulo de BCP é o modulo que mais precisa interagir e modificar o estado do resolvedor. Ele manipula a estrutura de valoração do solver removendo ou não cláusulas, podendo atribuir valores para as variáveis internas e até mesmo alterando a estrutura da fórmula CNF.

A principal justificativa para deixar esse módulo o mais livre possível é no crescente esforço que pesquisadores fazem para melhorar a técnica do BCP, que representa cerca de 90 % do tempo de execução [2] de um resolvedor. Ter um mecanismo BCP extremamente extensível possibilita dar suporte para esse tipo de experimento, sem a necessidade de implementar a lógica completa de um resolvedor.

Neste módulo é possível implementar o mecanismo de aprendizado de cláusulas de um resolvedor, além do mecanismos de propagação unitária. No Código 13, temos um exemplo de implementação simples do mecanismo de BCP, inicialmente o módulo busca a lista de cláusulas ligadas ao literal por meio da interface *getLiteralClauses*. Essa lista de cláusulas é uma lista com apenas uma ligação para o próximo nó da lista, o primeiro nó é retornado pela função. Vale notar que assim que como foi detalhado na seção de representação da sentença, não é uma boa ideia modificar essas estruturas sem utilizar a API, a lista original é retornada por questões de performance, fica a cargo do usuário saber lidar com essa interface apropriadamente.

4.4.4 Resolução de conflito

A resolução de conflitos é feita através do processo de retrocesso de decisões feita pelo resolvedor, uma vez que o módulo de BCP identifique um conflito é responsabilidade do módulo de resolução de conflitos decidir a ação que será tomada.

No Código 14 é ilustrado o processo de retrocesso feito pelo resolvedor, o objetivo deste módulo é indicar o nível em que existe uma decisão que pode ser alterada para que o Código 9 continue sua busca corretamente. Caso este módulo retorne o valor 0(zero) isso significa que o resolvedor não consegue se recuperar do conflito e pelo Código 9 a sentença será considerada UNSAT.

Ainda no Código 14, temos o uso de algumas APIs para manipulação da estrutura de decisão. A primeira interface *getDecisions* é responsável por retornar a estrutura de decisão, que é um vetor estático com o mesmo tamanho do número de variáveis, enquanto a interface *getLevel* retorna um inteiro que sinaliza quantas decisões foram feitas até o momento nesta estrutura.

```
bool BCP(Form *formula, const Decision decision){
    bool flag;
    ClauseNode *head;
    Clause *clause;

    LiteralId falseValuedLiteral = ((decision.value == FALSE) ?
                decision.id+1 : -decision.id -1);

    head = getLiteralClauses(falseValuedLiteral);

    while(head!=NULL){
        flag = false;
        clause = head->clause;

        for(int i = 0; i<clause->size; ++i){
            LiteralId lit = clause->literals[i];
            if(getLitState(lit) != FALSE)flag=true;
        }

        if(!flag) return false;
        head = head->next;
   }

   return true;
}
```

Código 13: Exemplo de implementação do módulo BCP

```
int resolveConflict()
{
    Decision* decisions =  getDecisions();

    int i = getLevel();

    for(; i>=0; --i)
        if(decisions[i].flipped == false)
            break;

    return i;
}
```

Código 14: Retrocesso e resolução de conflitos

4.5 Implementação

Resolvedores SAT completos tem requisitos que precisam ser atendidos para satisfazer sua funcionalidade. Em resolvedores SAT temos alguns requisitos: o uso de memória deve ser eficiente (principalmente em resolvedores com aprendizado), o resolver precisa encontrar uma solução independente do tempo gasto e o tempo para encontrar a solução precisa ser o menor possível.

Em um sistema que pode ser estendido pelo usuário, não é possível garantir que seu código atenda a todos esses requisitos sempre. Assim a implementação do PluSAT procura otimizar sua estrutura interna e algumas decisões de implementação ajudam o usuário a continuar atendendo esses requisitos.

4.5.1 Codificação do PluSAT

O PluSAT é escrito na linguagem C. Essa linguagem é bastante utilizada em resolvedores por conta do controle de memória que ela proporciona, resolvedores SAT precisam gerenciar cuidadosamente sua memória pois praticamente todas as ações adicionam uma carga na memória.

Além disso, para garantir que o sistema interno limpe corretamente toda memória alocada, o PluSAT utiliza programas auxiliares chamados de *sanitizers* que analisa se toda memória alocada foi desalocada no final do programa, diminuindo a chance de vazamento de memórias.

4.5.2 Mecanismo de *Plugins*

O mecanismo de extensão da estrutura externa são implementados como *Shared Objects* no sistema operacional. Sistemas operacionais modernos utilizam *Shared Objects* para manter versões de programas que são referenciados por outros programas, através do processo de *linking* dinâmico o sistema operacional é capaz de em tempo de execução encontrar determinada dependência e agrega-lá ao código sem a necessidade do código principal ter uma versão do binário.

Para implementarmos uma estrutura de *plugins* utilizando *Shared Object*, abrimos mão do *linking* dinâmico do sistema operacional e carregamos a implementação do usuário em tempo de execução. A diferença desse processo para o *linking* dinâmico, é que o programa em si é responsável por encontrar o objeto. No Código 15 representamos a base de uma implementação do usuário.

Com essa implementação em mãos criamos um *Shared Object* utilizando o compilador da linguagem C como exemplificado no Código 16. No repositório oficial do PluSAT, existem algumas automações que auxiliam neste processo, vale chamar atenção que o

```c
#include "dpll.h"
#include "formula.h"

void PreProcessing(Form* form){
        /* User Code */
}

enum DecideState Decide(const Form* form){
        /* User Code */
}

bool BCP(Form *formula, const Decision decision){
        /* User Code */
}

bool resolveConflict(){
        /* User Code */
}
```

Código 15: Exemplo de código implementado pelo usuário

```
gcc -fPIC -shared -o libuser.so userfile.c -I "dpll.h" -I "formula.h"
```

Código 16: Instrução para compilação de *Shared Object*

processo demonstrado assume que o usuário está compilando os arquivos diretamente no repositório principal, empacotamentos deste software para diversas distruições podem deixa esse processo mais simples.

Os exemplos e instruções que foram definidos até aqui foram feitos com base do repositório do PluSAT, porém, não é necessário ter todos os arquivos deste repositório para gerar o *plugin*. Toda a interface do usuário está definida e dividida em dois arquivos de definições chamados dpll.h e formula.h assim como mostrado no Código 16, apenas esses arquivos são necessários para gerar o *Shared Object* do *plugin*. É preciso implementar todas as interfaces do Código 15 para que o *plugin* funcione corretamente.

5 Conclusão

Neste trabalho verificamos a viabilidade de se construir um resolvedor SAT modulável. Provamos que é possível implementar uma variedade de algoritmos presentes em um resolvedor CDCL nesta estrutura e mais importante, que é possível abstrair parte da lógica de implementação de um resolvedor, o que é uma qualidade bastante interessante para pesquisadores e estudantes do problema.

Ainda existem áreas de melhorias dentro do Plusat. Criar uma camada de intermediação para alocações de objetos no PluSAT pode ajudar a identificar problemas de alocação e determinar com precisão vazamentos de memória, além de podermos alocar todo o espaço necessário na memória antes da execução do resolvedor diminuindo o tempo gasto com alocação. Adicionar mais implementações básicas que o usuário possa apenas utilizar, isso ajuda nos primeiros testes com o sistema e também diminui a necessidade de implementação de todos os componentes caso o interesse esteja em apenas um único específico. Empacotar o PluSAT para que não seja necessário compilar o código fonte diretamente.

Referências

1 COOK, S. A. The complexity of theorem-proving procedures. In: *Proceedings of the third annual ACM symposium on Theory of computing*. [S.l.: s.n.], 1971. p. 151–158. Citado na página 11.

2 MOSKEWICZ, M. W. et al. Chaff: Engineering an efficient sat solver. In: *Proceedings of the 38th annual Design Automation Conference*. [S.l.: s.n.], 2001. p. 530–535. Citado 4 vezes nas páginas 11, 19, 20 e 34.

3 EÉN, N.; SÖRENSSON, N. An extensible sat-solver. In: SPRINGER. *International conference on theory and applications of satisfiability testing*. [S.l.], 2003. p. 502–518. Citado 2 vezes nas páginas 11 e 15.

4 AUDEMARD, G.; SIMON, L. On the glucose sat solver. *International Journal on Artificial Intelligence Tools*, World Scientific, v. 27, n. 01, p. 1840001, 2018. Citado 2 vezes nas páginas 11 e 15.

5 RIBAS, B. C. Satisfatibilidade não-clausal restrita às variáveis de entrada. 2011. Citado 2 vezes nas páginas 15 e 19.

6 HOOS, H. H.; STTZLE, T. Satisfiability suggested format. *IP Gent, Hv Maaren, T. Walsh, editors, SAT 2000*, p. 283–292, 2000. Citado na página 15.

7 DAVIS, M.; PUTNAM, H. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, ACM New York, NY, USA, v. 7, n. 3, p. 201–215, 1960. Citado 2 vezes nas páginas 16 e 18.

8 BIERE, A.; HEULE, M.; MAAREN, H. van. *Handbook of satisfiability*. [S.l.]: IOS press, 2009. v. 185. Citado 5 vezes nas páginas 16, 17, 18, 19 e 20.

Apêndices

APÊNDICE A – Sumário funções PluSAT

Função	Descrição
Clause* newClause(LiteralId*, uint8_t)	Cria uma nova cláusula, não ligada à fórmula
void freeClause(Clause*)	Limpa a cláusula da memória (não necessário caso a cláusula esteja ligada a fórmula)
ClauseNode* addNodeOnList(Clause*, ClauseNode*)	Adiciona uma cláusula em uma lista de cláusulas
void freeList(ClauseNode*)	Limpa uma lista de cláusulas
Form* newForm(uint16_t)	Cria uma nova fórmula na memória, como argumento recebe o número de variáveis desta fórmula
void freeForm(Form*)	Limpa a fórmula da memória e todas as suas estruturas ligadas (cláusulas, literais e tabelas de literais)
ClauseNode* getLiteralClauses(const LiteralId, const Form*)	Busca a lista de cláusulas ligada a um literal da fórmula
void addClause(Clause*, Form*)	Adiciona uma cláusula a uma fórmula
int getLevel()	Retorna o nível na pilha de decisão
void insertDecisionLevel(const VariableId, const int)	Adiciona uma decisão para uma determinada variável
void backtrackTo(uint16_t)	Retorna para o nível de decisão enviado para essa função
Decision* getLastDecision()	Retorna uma referência da última decisão feita
enum LiteralStates getLitState(LiteralId)	Retorna o estado de um literal, dentre as opções existem: UNK, TRUE e FALSE
void setVarState(VariableId, LitState)	Escolhe um valor para uma variável sem realizar uma decisão, apenas assina um LitState a ela.
LitState getVarState(const VariableId var)	Retorna o valor de uma variável
Decision* getDecisions()	Retorna uma referência ao vetor de decisões do resolvedor, esse vetor tem o mesmo tamanho do número de variáveis de uma fórmula