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

Representação AIG como formato de entrada para o LIAMFSAT

Autor: Lucas Gomes Silva
Orientador: Prof. Dr. Bruno César Ribas

Brasília, DF
2022

Lucas Gomes Silva

Representação AIG como formato de entrada para o LIAMFSAT

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

Lucas Gomes Silva

Representação AIG como formato de entrada para o LIAMFSAT

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, 06 de maio de 2022.

Prof. Dr. Bruno César Ribas
Orientador

Profa. Dra. Cláudia Nalon
Convidado 1

Prof. Dr. Fabricio Braz
Convidado 2

Brasília, DF
2022

Agradecimentos

A Deus, por ter me dado forças para chegar até aqui. Aos meus pais e irmãs, que sempre me apoiaram e deram forças em todas as escolhas que realizei. A minha namorada, Karine, que nunca poupou esforços para estar ao meu lado e me apoiar. Aos meus amigos, Carlos, Júlia, Thamiris, Giovanna, Adriely e Gabriel, por compreenderem as minhas ausências para me dedicar aos estudos, além de me ajudarem sempre que necessitei. Ao meu amigo e colega de curso, Julio, por ter trilhado ao meu lado o período de graduação. Ao Professor Bruno Ribas, por todo conhecimento passado não só neste trabalho, mas em toda a graduação.

Resumo

A Satisfatibilidade Booleana (SAT) é uma abordagem amplamente utilizada para verificação de *software* e *hardware*. Existem diversos resolvedores voltados para a resolução do problema de satisfatibilidade baseados no algoritmo Davis-Putnam-Logemann-Loveland (DPLL) e precisam que a fórmula esteja na Forma Normal Conjuntiva (CNF). Porém, muitos problemas não se encontram na representação CNF e então é necessária uma abordagem diferente para a resolução destes problemas. Os resolvedores não-clausais são voltados para a resolução de problemas que não se encontram na representação CNF, isto é, se encontram na representação não-clausal. Resolvedores não-clausais possuem diferentes abordagens, desde a conversão da fórmula não-clausal para a representação CNF à atuação diretamente na fórmula. Entretanto, a conversão para CNF pode ocasionar na perda considerável de informações sobre a estrutura do problema. Portanto, o objetivo deste trabalho consiste em um novo formato de entrada no resolvedor não-clausal LIAMFSAT, em que a abordagem utilizada foi a implementação da representação Grafo AND-Inversor (AIG), além de melhorias no resolvedor de modo a de facilitar trabalhos futuros.

Palavras-chave: lógica proposicional. satisfatibilidade. resolvedor não-clausal. AIGER.

Abstract

Boolean Satisfiability (SAT) is a widely used approach for checking software and hardware. There are several solvers aimed at solving the satisfiability problem based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and need the formula to be in the Conjunctive Normal Form (CNF). However, many problems are not in the CNF representation, so a different approach to solve these problems is needed. Non-clausal solvers are aimed to solve problems that are not in the CNF representation, that is, the problems are in the non-clausal representation. Non-clausal solvers have different approaches, from converting the non-clausal formula to the CNF representation to acting directly on the formula. However, the conversion to CNF can cause considerable loss of information about the structure of the problem. Therefore, the main objective of this work is a new input format to the LIAMFSAT non-clausal solver, in which the approach used was the implementation of the Graph AND-Inverter (AIG) representation, as well as improvements to the solver in order to facilitate future works.

Key-words: propositional logic. satisfiability. non-clausal solver. AIGER.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

CNF	Conjuctive Normal Form
DAG	Directed Acyclic Graph
AIG	And-Inverter Graph
SAT	Satisfatibilidade
DPLL	Davis-Putnam-Logemann-Loveland
ASCII	American Standard Code for Information Interchange
LHS	Left-hand side
RHS	Right-hand side
CSGR	Circuit Sat Gomes Ribas
LIAMF	Laboratório de Inteligência Artificial e Métodos Formais
UFPR	Universidade Federal do Paraná
BCP	Boolean Constraint Propagation
DP	Davis-Putnam
DPLL	Davis-Putnam-Logemann–Loveland
GPU	Graphics Processing Unit
CPU	Central Processing Unit

Lista de símbolos

$\wedge$	Conjunção
$\vee$	Disjunção
$\neg$	Negação
$\oplus$	Disjunção exclusiva
$\overline{\wedge}$	Conectivo de Sheffer
$\Rightarrow$	Implicação
$\Leftrightarrow$	Equivalência
α	Alfa
β	Beta
γ / Γ	Gama
Δ	Delta

Sumário

1 Introdução

Segundo Laplante e DeFranco (2017), muito tem se aplicado sistemas computacionais em aplicações críticas de diferentes setores. Para Ribas (2011), é necessário garantir que o funcionamento de uma aplicação crítica ocorra de forma correta. Um exemplo de mau funcionamento de uma aplicação crítica foi a explosão de um gasoduto na cidade de Bellingham no ano de 1999 que ocasionou na morte de três pessoas (MCCLARY, 2003). Mas, verificar se um sistema está funcionando corretamente tem sua complexidade proporcional ao tamanho do sistema, isto é, a complexidade de verificação aumenta a medida que o sistema cresce. Desta forma, a verificação automática do *software* e *hardware* se torna cada vez mais necessária (RIBAS, 2011).

Ribas (2011) relata que "muitas ferramentas de verificação automática dependem de procedimentos de decisão para verificar a satisfatibilidade de várias fórmulas lógicas proposicionais geradas durante o processo de verificação". Com as várias fórmulas geradas é necessária uma forma eficaz para verificar a satisfatibilidade delas, de forma que não seja grande o tempo de verificação já que uma fórmula pode possuir uma ampla quantidade de variáveis e operadores.

Com as fórmulas lógicas proposicionais retornadas da verificação automática e a necessidade de verificar a satisfatibilidade da fórmula é então necessária uma abordagem que analise a fórmula lógica de forma que a torne verdadeira. Diante da necessidade de saber se uma fórmula é satisfatível ou não, o problema da Satisfatibilidade Booleana (SAT) encontra uma valoração para as variáveis da fórmula que a torne verdadeira, isto para quando a fórmula é satisfatível. O interesse em SAT tem crescido cada vez mais e, provavelmente, isto se deve ao fato de a satisfatibilidade envolver diversas áreas, como lógica, teoria dos grafos, ciência da computação, engenharia da computação e pesquisa operacional (FRANCO; MARTIN, 2009). Segundo Cook (1971), "SAT foi o primeiro problema provado NP-Completo e nenhum algoritmo eficiente é conhecido para resolvê-lo".

Existem ferramentas com a capacidade de analisar uma fórmula lógica e dizer se é satisfatível ou não de forma automática, estas aplicações são os chamados resolvedores SAT. Ao longo dos anos, os resolvedores tem deixado cada vez mais sua marca como aplicação de uso geral devido à crescente variedade de áreas que possibilitam a sua utilização. Tem-se como exemplo as áreas de teste automatizado de programas (LINGAPPAN; JHA, 2007), diagnóstico de falhas (SMITH et al., 2005), planejamento (RUSSELL; NORVIG; CHANG, 2010), depuração automática de sistemas em tempo real (ANDREI et al., 2006), sistemas biológicos (LYNCE; MARQUES-SILVA, 2006) e diversos outros estudos

em engenharia da computação em geral (KHURSHID; MARINOV, 2004). A existência de competições voltadas para o problema de satisfatibilidade tem influenciado ainda mais o desenvolvimento de implementações mais sofisticadas dos resolvedores, como a exploração de novas técnicas e a criação de diversos problemas reais (GOMES et al., 2007).

É descrito por Darwiche e Pipatsrisawat (2009) que muitos resolvedores SAT precisam que a fórmula de entrada esteja na Forma Normal Conjuntiva (CNF), ou seja, representada pela conjunção de disjunções. Porém, muitas fórmulas do meio real não se encontram no formato CNF e estas fórmulas são chamadas não-clausais.

Fórmulas não-clausais podem ser abordadas de diferentes formas onde é possível a sua conversão para CNF ou atuar diretamente na representação original. A atuação direta na fórmula original possibilita a utilização de heurísticas específicas com base na informação estrutural da fórmula, como em caso de circuitos onde operadores podem receber prioridade de valoração em relação aos demais (DRECHSLER; JUNTTILA; NIEMELA, 2009).

Desta forma, este trabalho tem por objetivo principal um novo formato de entrada no resolvedor não-clausal LIAMFSAT, em que a abordagem utilizada foi a implementação da representação Grafo AND-Inversor (AIG). Foram definidos também objetivos específicos, são eles:

- Restaurar um código antigo;
- Realizar melhorias de modo a facilitar trabalhos futuros;
- Realizar experimentos de desempenho.

Este trabalho inicia pelo embasamento teórico no Capítulo 2, necessário a compreensão dessa teoria para assim ser possível a implementação da proposta deste trabalho. O Capítulo 3 apresenta o funcionamento de um resolvedor não-clausal e alguns recursos utilizados no processo de resolução do problema de satisfatibilidade aplicado a uma fórmula. O resolvedor não-clausal selecionado para este trabalho foi o LIAMFSAT para dar continuidade ao desenvolvimento realizado pelo orientador deste trabalho, Bruno César Ribas. No Capítulo 4 é relatado o trabalho realizado no resolvedor não-clausal escolhido e no Capítulo 5 é apresentado experimentos executados após implementação das alterações citadas no Capítulo 4. Por fim, o Capítulo 6 finaliza este trabalho apresentando considerações sobre o processo de desenvolvimento do mesmo e ideias para trabalhos futuros.

2 Referencial Teórico

Neste Capítulo é exposto o embasamento teórico necessário para a implementação deste trabalho. É apresentado o conceito referente a Lógica Proposicional e suas diferentes formas de representação, o Problema da Satisfatibilidade Booleana e o formato AIGER.

2.1 Lógica Proposicional

A lógica proposicional, também conhecida como lógica sentencial ou lógica declarativa, é um ramo da lógica em que se estuda formas de combinar ou alterar declarações, ou proposições, para formar declarações mais complexas. Uma *proposição* pode ser definida como um fato e pode ser representada como uma sentença declarativa, ou parte de uma sentença, em que é possível atribuir um valor de verdade, seja ele verdadeiro ou falso, mas nunca ambos na lógica clássica. Além da representação por uma sentença declarativa, a proposição pode ser representada também por fórmulas na linguagem da lógica escolhida. Um exemplo de proposição é "Paris é a capital da França". Às vezes, uma proposição pode conter uma ou mais proposições em sua composição, por exemplo, "Ou Ganimedes é uma lua de Júpiter ou Ganimedes é uma lua de Saturno" (KLEMENT, 2004).

Uma fórmula é construída através de átomos (também chamados *variáveis proposicionais* ou *proposições elementares*), onde representam proposições básicas e podem receber apenas o valor *verdadeiro* ou *falso*. Utilizando operadores lógicos (conectivos) para combinar átomos é possível criar fórmulas complexas. Os operadores lógicos disponíveis para serem utilizados são o de conjunção ('e' lógico, denotado por $\wedge$), disjunção ('ou' lógico, denotado por $\vee$), negação ('não' lógico, denotado por $\neg$), implicação (se ... então, denotado por $\Rightarrow$) e equivalência (se e somente se, denotado por $\Leftrightarrow$) (BÜNING et al., 1999).

Büning et al. (1999) e Klement (2004) definem algumas notações básicas para a lógica proposicional, são elas:

- Fórmulas são denotadas por metavariáveis usando letras gregas como, por exemplo, Δ, α e γ.
- *Variáveis proposicionais* são denotados como x_1, ..., x_n ou por letras do alfabeto em minúsculas como p, q, r e podem receber valores de verdade *verdadeiro* ou *falso*.
- O valor de verdade *verdadeiro* pode ser representado pela letra V e valor *falso* pela letra F.

- O valor de verdade assinalado em uma variável x é denotado por $v : X \to \{V, F\}$, onde X é um conjunto finito de variáveis e $x \in X$.
- Um *literal* l é uma variável x ou a sua negação $\neg x$.
- Uma *Cláusula Unitária* é uma cláusula com apenas um literal.
- *Literal Puro* é o literal que aparece em apenas uma forma em toda a fórmula, ou seja, a variável x aparece apenas na forma x ou $\neg x$ na fórmula.
- Um conectivo, ou operador lógico, é qualquer um dos sinais $\neg$, $\wedge$, $\vee$, $\Rightarrow$ e $\Leftrightarrow$.

Para Büning et al. (1999), uma fórmula pode ser definida pelas seguintes quatro regras:

1. Todo átomo é uma fórmula.
2. Se Δ é uma fórmula, então $(\neg\Delta)$ também é uma fórmula.
3. Se Δ e α são fórmulas, então $(\Delta \wedge \alpha)$, $(\Delta \vee \alpha)$, $(\Delta \Rightarrow \alpha)$ e $(\Delta \Leftrightarrow \alpha)$ também são fórmulas.
4. Todas as fórmulas são geradas pelas aplicação das regras anteriores.

2.2 Formas de representação

As fórmulas proposicionais podem ser representadas na forma de um grafo direcionado acíclico (DAG), mas possui algumas diferenças se comparado a sua forma escrita. Ao utilizar grafo como formato de representação, há uma diminuição da repetição de literais representados devido que cada literal possui o seu próprio vértice. Outra diferença a ser citada é o fato da representação da negação passar por dois vértices, sendo uma negação e o outro o literal (WANG; DING, 2013).

Primeiramente uma breve explicação sobre algumas formas de representação utilizando grafos, como a Forma Normal Conjuntiva e algumas formas não-clausais.

2.2.1 Forma Normal Conjuntiva

A Forma Normal Conjuntiva, em inglês *Conjuctive Normal Form*, também chamada por forma clausal, é uma conjunção de cláusulas. Cada cláusula é uma disjunção de literais, isto é, os literais são conectados através de um operador lógico de disjunção para formar uma cláusula, exceto nos casos em que a disjunção tem tamanho zero ou

um. As cláusulas são conectadas por um operador lógico de conjunção para gerar a fórmula proposicional. Deste modo, os únicos operadores lógicos utilizados nessa forma é a conjunção, disjunção e negação (WARNERS, 1996).

A Figura 1 ilustra como é representada a fórmula $(p \vee q) \wedge (q \vee \neg p) \wedge (\neg p \vee \neg q)$ em formato de grafo.

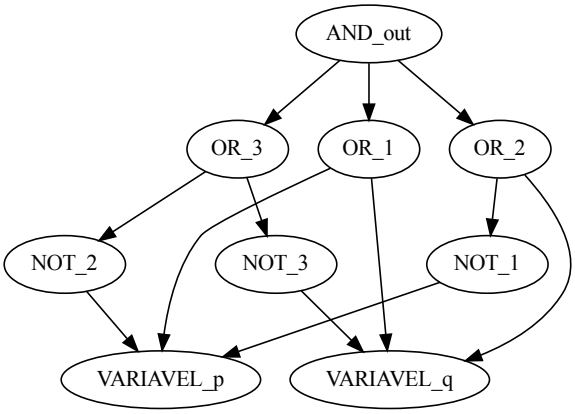

Figura 1: Diagrama de uma fórmula CNF. Fonte: Autor.

A figura anterior representa em formato de grafo uma fórmula CNF em que a raiz é um operador de conjunção, no próximo nível é constituído por operadores de disjunção e, por fim, as folhas são compostas pelos literais, formando assim um grafo de largura 3.

2.3 Representações não-clausais

Além da representação CNF, existem outras diversas formas de representação, as não-clausais. As representações não-clausais são as fórmulas em que não estão no formato CNF (DRECHSLER; JUNTTILA; NIEMELA, 2009).

Há a possibilidade da conversão da representação não-clausal para a CNF, mas Thiffault, Bacchus e Walsh (2004) relatam que esta conversão é desnecessária. E segundo os autores já mencionados, mas citados por Jain e Clarke (2009), realizar a conversão de uma fórmula não-clausal em CNF pode acarretar perda de informação estrutural da fórmula, mesmo que não cause uma explosão combinatorial no seu tamanho, prejudicando a verificação da satisfatibilidade.

2.3.1 ISCAS

Não foi encontrada documentação formal para esta representação, assim como as outras, mas ela é bastante utilizada em circuitos eletrônicos e, por esta razão, pode haver

mais de uma saída, além da possibilidade do uso de diversos operadores lógicos, por exemplo, $\wedge$, $\vee$, $\neg$, $\oplus$ e $\overline{\wedge}$ (BRYAN, 1985).

Neste formato não há nenhuma restrição de operadores, logo não há um limite para a quantidade de filhos e pais, com exceção do $\neg$ que pode ter apenas um filho. Em seu formato de grafo, os nós folhas podem receber verdadeiro, falso ou não valorado e cada nó interno pode receber qualquer um dos diversos operadores lógicos disponíveis (RIBAS, 2011).

A Figura 2 ilustra como é representado a fórmula

$$((p \vee q) \wedge (r \vee \neg s) \wedge \neg q) \vee \neg(\neg p \wedge (r \vee \neg s) \wedge q) \tag{2.1}$$

em formato de grafo.

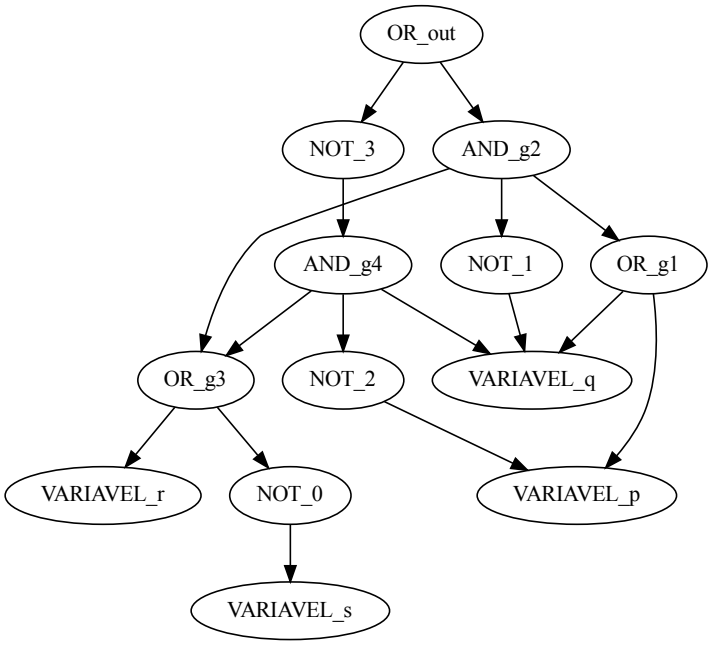

Figura 2: Diagrama de uma fórmula ISCAS. Fonte: Autor.

2.3.2 EDIMACS

Para Bacchus e Walsh (2005), o formato ISCAS já fornecia uma boa base para fórmulas não clausais, mas nunca foi documentada formalmente, então o EDIMACS foi criado para encorajar desenvolvedores de resolvedores SAT e para auxiliar uma competição não-clausal na conferência anual de SAT.

Ainda conforme os autores citados, a fórmula neste formato é especificada como uma coleção de operadores lógicos que em um determinado momento chegará a uma única

saída raiz e segue a mesma forma de representação do CNF. O EDIMACS possui diversos operadores lógicos compatíveis, muitos deles presentes no ISCAS. A Figura 3 ilustra como é representada a fórmula $(p \vee q) \wedge (q \vee r) \wedge (r \vee (p \wedge s)) \wedge (t \vee (s \wedge u))$ em formato de grafo.

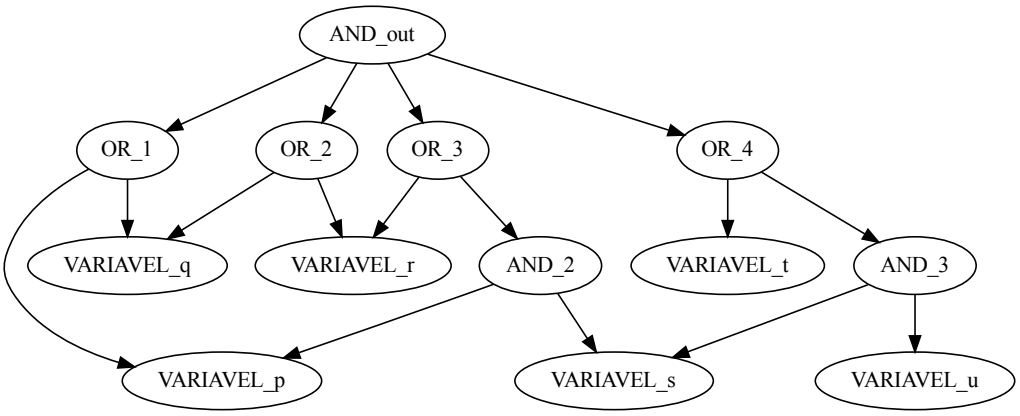

Figura 3: Diagrama de uma fórmula no formato EDIMACS. Fonte: Autor.

2.3.3 AIG

Segundo Brummayer e Biere (2006, tradução nossa), "os grafos AND-inversor (AIGs) são uma eficiente e escalável representação para fórmulas Booleana e circuitos". E Kuehlmann e Krohm, citados por Neto (2018), relatam que inicialmente o AIG era utilizado para verificação de equivalência combinacional e é uma estrutura de dados muito utilizada para realização de síntese lógica.

De acordo com Mishchenko, Chatterjee e Brayton (2006), este formato possui as seguintes características:

- Os nós possuem nenhuma ou duas arestas de entrada;
- Nós que não possuem arestas de entrada são nós de entrada;
- Nós de entrada são as variáveis da fórmula;
- Nós que possuem duas arestas de entrada são nós de conjunção ($\wedge$);
- Uma aresta pode ser complementada ou não;
- Aresta complementada indica a inversão do valor lógico;
- Nós de saída são nós que não possuem arestas de saída para outros nós;
- O nó de saída indica se a fórmula é verdadeira ou falsa.

Baseado na fórmula

$$\neg(\neg(p \wedge q) \wedge (\neg(q \wedge r))), \tag{2.2}$$

a Figura 4 ilustra como é a sua representação em formato de grafo.

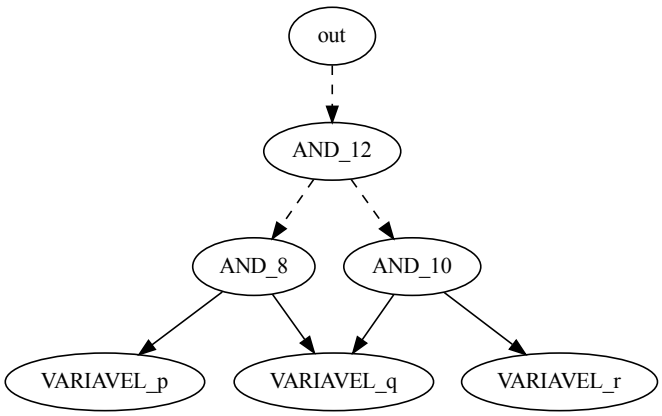

Figura 4: Diagrama de uma fórmula no formato AIG. As arestas complementadas, onde ocorrem a inversão de valor, são representadas pelas linhas tracejadas. Fonte: Autor.

A figura anterior foi representada desta forma, com as arestas apontando para os filhos, devido à representação usual das outras figuras e com o intuito de indicar que os nós pais sabem quem são os nós filhos, mas os filhos não sabem quem são os pais. Entretanto, não seria incorreto as arestas apontarem para os pais e representando desta forma estaria retratando a descrição dada por Mishchenko, Chatterjee e Brayton (2006).

2.4 Satisfatibilidade e resolvedores de satisfatibilidade

O problema da satisfabilidade Booleana (SAT) é um problema NP-completo que consiste em decidir se uma determinada fórmula é satisfatível ou não (FRANCO; MARTIN, 2009).

Para resolver o problema de satisfatibilidade são desenvolvidos programas, os resolvedores SAT, que verificam de forma automática se uma fórmula é satisfatível ou não. Geralmente o formato CNF é utilizado nesses resolvedores devido à sua simplicidade e utilidade (GOMES et al., 2007).

Os resolvedores já desenvolvidos possuem diversas abordagens e algoritmos diferentes, como a conversão de uma fórmula não-clausal para CNF e atuação diretamente na fórmula não-clausal (THIFFAULT; BACCHUS; WALSH, 2004). Mouhoub e Sadaoui

(2007) apresentam algumas abordagens utilizadas em resolvedores, como técnicas de busca sistemática, busca local estocástica e algoritmos genéticos. E, mesmo com a crescente evolução dos algoritmos implementados nos resolvedores, os melhores algoritmos completos permanecem variantes de um algoritmo introduzido há várias décadas, o Davis-Putnam-Logemann-Loveland (DPLL) (GOMES et al., 2007).

O DPLL é um algoritmo baseado em *backtracking* que executa buscas sistemáticas no espaço das atribuições de valores de verdade. Este algoritmo é famoso por sua baixa exigência de espaço e foi desenvolvido em resposta aos requisitos de espaço do algoritmo de Davis-Putnam (DARWICHE; PIPATSRISAWAT, 2009). Segundo Gomes et al. (2007, tradução nossa), "desde a sua introdução no início dos anos 1960, as principais melhorias no DPLL foram heurísticas de seleção de ramificação inteligente, extensões como aprendizagem de cláusula e reinicialização aleatória, e estruturas de dados bem elaborados como implementações preguiçosas e literais observados para propagação rápida de unidades".

Com o avanço da área de resolvedores SAT, muitos problemas, que até anos atrás pareciam estar completamente fora de alcance, podem ser tratados rotineiramente. Com isso, foram organizadas diversas competições que ocorrem anualmente em diferentes países visando criar experimentos cada vez mais desafiadores e incentivar a criação de novos resolvedores para o problema de satisfatibilidade, bem como compará-los com solucionadores de última geração. Estas competições visam também a criação de novos algoritmos, melhores heurísticas e técnicas de implementação refinadas (SATCompetitions, 2005).

2.4.1 Algoritmo DPLL

Segundo Davis e Putnam (1960), o algoritmo DP possui três regras. Considerando uma fórmula Δ e seu conjunto de cláusulas α, as regras são as seguintes:

1. **Cláusula Unitária:** se existe alguma cláusula unitária em α dada pelo literal x, remova de α todas as cláusulas contendo x e todas as ocorrências do literal $\neg x$ das cláusulas. Se α estiver vazia após a remoção, então Δ é satisfatível;

2. **Literal Puro:** se o literal x estiver em alguma cláusula em α, mas o literal $\neg x$ não estiver, remova todas as cláusulas que contenham x. Neste caso, o literal x é chamado de *literal puro*. Se α estiver vazia após a remoção, então Δ é satisfatível;

3. **Eliminando fórmulas atômicas:** considerando que $(p \vee q) \wedge (\neg p \vee r) \wedge s$ seja a fórmula Δ e q, r e s são livres de p, é possível remover as ocorrências das variáveis p e/ou $\neg p$ ao agrupar as cláusulas que a possuem. Ao realizar estas ações na fórmula Δ, ela se tornará a fórmula $(q \vee r) \wedge s$. Desta forma, a fórmula Δ será insatisfatível se e somente se a fórmula $(q \vee r) \wedge s$ também for insatisfatível.

Entretanto, Davis, Logemann e Loveland (1962) ao programar o algoritmo DP perceberam que poderia facilmente aumentar o número e o tamanho das cláusulas na fórmula rapidamente, além do aparecimento de muitas cláusulas repetidas, devido a regra 3. Assim, propuseram uma nova regra para substitui-la e esta regra diz que:

- **Regra de Divisão:** considerando que $(p \vee q) \wedge (\neg p \vee r) \wedge s$ seja a fórmula Δ e q, r e s são livres de p. A fórmula Δ será insatisfatível se e somente se $q \wedge s$ e $r \wedge s$ forem ambas insatisfatíveis.

Assim, uma variável seria valorada recursivamente com *verdadeiro* e $falso$ e ambos subproblemas resultantes seriam resolvidos. Esta alteração originou o algoritmo DPLL (FRANCO; MARTIN, 2009). O Algoritmo 1 descreve o procedimento DPLL.

Entrada: Fórmula Δ em CNF
Saída: conjunto de literais ou INSATISFATIVEL

```
se Δ = {} então
    retorna {}
fim
senão se {} ∈ Δ então
    retorna INSATISFATIVEL
fim
senão
    u ← algum literal ∈ Δ
    se α ← DPLL(Δ|u) ≠ INSATISFATIVEL então
        retorna α ∪ {u}
    fim
    senão se α ← DPLL(Δ|¬u) ≠ INSATISFATIVEL então
        retorna α ∪ {¬u}
    fim
    senão
        retorna INSATISFATIVEL
    fim
fim
```

Algoritmo 1: Algoritmo DPLL. Fonte: Darwiche e Pipatsrisawat (2009).

Segundo Marques-Silva, Lynce e Malik (2009), a aplicação iterada da regra da Cláusula Unitária é chamada *Boolean Constraint Propagation (BCP)*, ou *Propagação Unitária*, usada para identificar variáveis que devem ser atribuídas um valor de verdade específico. Na próxima seção é descrito o processo de propagação através da valoração de uma variável e o seu funcionamento no algoritmo DPLL.

2.4.2 *Boolean Constraint Propagation* (BCP)

O *Boolean Constraint Propagation* (BCP) é responsável por identificar as implicações de valoração e propagar o efeito da valoração feita até o momento, então o algoritmo procura alguma cláusula que ficou falsa, ou retorna as implicações causadas pela última decisão (RIBAS, 2011). De acordo com Moskewicz et al. (2001), essa verificação é realizada em torno da valoração parcial sempre que o BCP é chamado, um dos procedimentos mais importantes e mais custosos do resolvedor, onde o seu consumo varia entre 80–90% do tempo gasto pelo resolvedor.

Darwiche e Pipatsrisawat (2009) explica que durante a propagação é realizado a operação de *condicionamento* que consiste na valoração de um literal e como esta valoração propaga pelo restante da fórmula. Esta operação remove da fórmula original cláusulas ou literais conforme o valor de verdade definido ao literal. Basicamente, o condicionamento de uma fórmula a um literal equivale em substituir todas as ocorrências do literal v pelo valor de verdade *verdadeiro*, do literal $\neg v$ por $falso$ e realizar as simplificações necessárias. O condicionamento da fórmula Δ em um literal v é denotado por $\Delta|v$ e pode ser descrita pela seguinte equação:

$$\Delta|v = \{\alpha - \{\neg v\} | \alpha \in \Delta, v \notin \alpha\}.$$

E, definido por Darwiche e Pipatsrisawat (2009), o processo de propagação realizado pela operação de condicionamento segue três regras, tendo como exemplo um literal $v \in \Delta$, são elas:

1. O valor de verdade do literal v é *verdadeiro*, assim todas as cláusulas que possuem o literal v se tornam satisfatíveis e podem ser removidas da fórmula Δ. Estas cláusulas não aparecem em $\Delta|v$.
2. O valor de verdade do literal $\neg v$ é $falso$ e todas as ocorrências de $\neg v$ em Δ são removidas, mas as cláusulas aparecem em $\Delta|v$ sem o literal $\neg v$.
3. As cláusulas que não possuem v ou $\neg v$ não sofrem alteração.

Com a definição do operador de condicionamento é possível realizar a propagação através de uma fórmula. Tendo como exemplo a fórmula

$$\Delta = \{\{\neg p, s\}, \{q, \neg s\}, \{p, r\}\} \tag{2.3}$$

e condicionando Δ ao literal s ($\Delta|s$) gerará a fórmula

$$\Delta|s = \{\{q\}, \{p, r\}\}$$

derivada de Δ.Algoritmo

O processo de propagação da Fórmula 2.3 é representado na Figura 5.

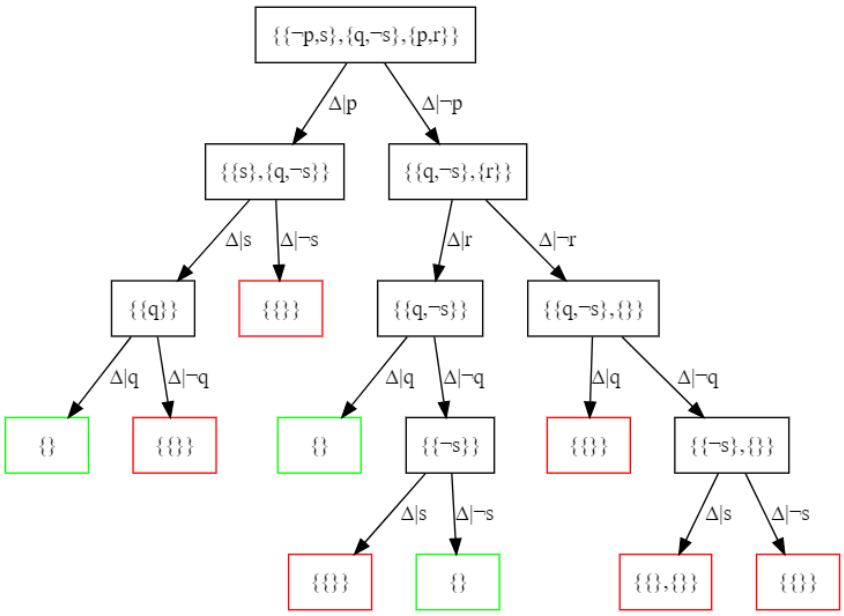

Figura 5: Propagação da Fórmula 2.3 a partir da variável p. Fonte: Autor.

A Figura 5 demonstra a propagação da valoração a partir dos literais p e $\neg p$. Em cada vértice é indicado a operação de condicionamento em Δ a um literal específico. Os nós que possuem a borda verde indica uma valoração satisfatível, enquanto a de borda vermelha indica insatisfatível.

Agora, a Figura 6 descreve uma árvore de busca com os valores de verdade atribuído as variáveis da fórmula 2.3.

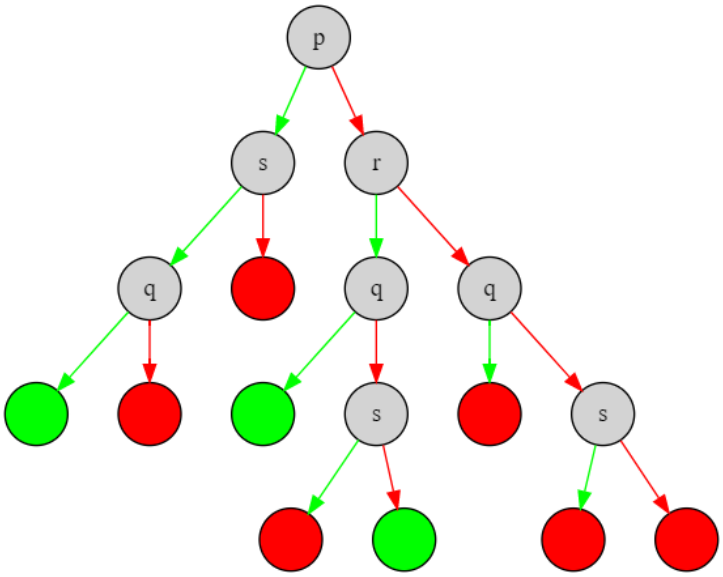

Figura 6: Árvore de busca com a propagação da fórmula 2.3 a partir da variável p. Fonte: Autor.

Na Figura 6, os vértices representam o valor de verdade atribuído a variável. Os vértices verdes representam o valor de verdade *verdadeiro* e os vértices vermelhos representam $falso$. Os nós folhas representam se o valor de verdade atribuído tornou a fórmula satisfatível ou insatisfatível através da cor do nó, verde para satisfatível e vermelho para insatisfatível.

Uma observação feita por Darwiche e Pipatsrisawat (2009) sobre a árvore de busca é que os nós folhas da árvore estão em um relacionamento um-para-um com um nó de variável que receberá o valor de verdade. Logo, para testar a satisfatibilidade de uma fórmula é possível realizar o processo de busca por um nó folha que satisfaça a fórmula dada.

Ao realizar o processo de propagação através do BCP, é verificado se possui alguma cláusula conflitante e caso seja identificada uma, uma condição de conflito é declarada e o algoritmo realiza o processo de retrocesso (DARWICHE; PIPATSRISAWAT, 2009). O processo de retrocesso será descrito na próxima seção.

2.4.3 Retrocesso não-cronológico e Aprendizagem de cláusula

Caso seja identificado uma cláusula conflitante durante a execução do BCP é realizado o processo de retrocesso. O algoritmo DPLL é baseado no *retrocesso cronológico*.

O retrocesso cronológico consiste em retroceder um nível de decisão caso ambos valores de verdade já tenham sido atribuídos a variável e sendo atribuído o valor de verdade restante a variável de nível $l-1$, onde l é o nível de decisão atual. Se todos os valores de verdades já foram atribuídos no nível $l-1$, é realizado outro retrocesso para $l-2$ e é realizado o mesmo processo de verificação se resta algum valor de verdade restante. Esse processo continua até encontrar uma variável valorada por apenas um valor de verdade ou até chegar ao nível 0. Caso o retrocesso chegue até o nível 0 e não tenha nenhum valor de verdade para atribuir a variável, a fórmula é insatisfatível. O processo de retrocesso desfaz todas as valorações realizadas naquele nível (DARWICHE; PIPATSRISAWAT, 2009).

É relatado por Ribas (2011) que mediante contribuições para a resolução do problema de satisfatibilidade foi introduzido um novo tipo de retrocesso no algoritmo DPLL. Este retrocesso se tornou mais inteligente, pois não é feito apenas a troca do valor de verdade da última decisão, o algoritmo passou a analisar o motivo da cláusula se tornar conflitante e retroceder todos os níveis até o nível de decisão que causou o conflito para tentar uma valoração diferente para a variável. Este processo é chamado de *retrocesso não-cronológico*.

Cada vez que o BCP identifica uma cláusula conflitante e adiciona essa cláusula a fórmula é uma forma do BCP detectar este mesmo conflito mais cedo e evitar o mesmo erro no futuro. O processo de adicionar a cláusula conflitante a fórmula é chamado de

aprendizagem de cláusula (DARWICHE; PIPATSRISAWAT, 2009).

Na Seção 3.2 possui um exemplo do processo de retrocesso não-cronológico e a aprendizagem de cláusula.

Assim, o Algoritmo 2 é descrito o algoritmo DPLL com a adição do BCP, retrocesso não-cronológico e aprendizagem de cláusula.

Entrada: Fórmula Δ em CNF
Saída: SATISFATIVEL ou INSATISFATIVEL
1 D $\leftarrow$ () //sequência de decisão vazia
2 $\Gamma \leftarrow$ () //conjunto vazio de cláusulas aprendidas
3 **enquanto** *verdadeiro* **faça**
4 **se** *BCP identifica um conflito em* (Δ, Γ, D) **então**
5 **se** $D = ()$ **então**
6 **retorna** *INSATISFATIVEL*
7 **senão**
8 //retrocesso até a valoração da cláusula conflitante
9 $\alpha \leftarrow$ cláusula que gerou o conflito
10 $m \leftarrow$ nível de decisão da cláusula α
11 D $\leftarrow$ primeiras m decisões em D //remove de D decisões a partir de de $m+1$
12 $\Gamma \leftarrow \Gamma \cup \alpha$ //α é adicionado a lista de cláusulas aprendidas Γ
13 **fim**
14 **senão**
15 **se** *l é um literal sem implicações pelo BCP* **então**
16 $D \leftarrow D + l$
17 **senão**
18 **retorna** *SATISFATIVEL*
19 **fim**
20 **fim**
21 **fim**

Algoritmo 2: Algoritmo DPLL com BCP, retrocesso não-cronológico e aprendizagem de cláusula. Fonte: Darwiche e Pipatsrisawat (2009).

2.5 AIGER

AIGER é um formato, biblioteca e um conjunto de utilitários voltado para AIGs em que foi introduzida por Armin Biere (BIERE, 2007). Em sua biblioteca é disponibilizado diversas ferramentas para auxiliar no trabalho com arquivos AIGER (BITTENCOURT,

2018).

Este formato até então possui duas versões, a primeira datada de 2007 e a segunda sendo de 2011. A versão de 2007 possui cinco diferentes propriedades em seu cabeçalho que ditam como será o conteúdo do arquivo, como índice máximo de variáveis, quantidade de entrada, *latches*, saídas e operadores $\wedge$ na fórmula (BIERE, 2007). Na versão de 2011 foram adicionados cinco novos recursos, como reinicialização lógica, restrições invariantes, propriedades de justiça, restrições de justiça e múltiplas propriedades. Entre as múltiplas propriedades adicionadas estão o número de estados ruins, restrições invariantes, propriedades de justiça e restrições de justiça. Com estas adições é possível a criação de circuitos mais complexos (BIERE; HELJANKO, 2011). Daremos uma ênfase maior a versão de 2007 devido o escopo deste trabalho.

O formato AIGER possui dois formatos diferentes de conteúdo, um em ASCII e o outro em binário. O formato em ASCII é simples de ser gerado e possui menos restrições, já em binário é o oposto, em que é mais restrito porém muito mais compacto (BIERE, 2007). A composição dos dois arquivos possuem algumas diferenças além do fato de um estar em ASCII e o outro binário. Estas diferenças serão explicadas mais adiante.

Biere (2007) desenvolveu o formato AIGER para que o arquivo possua um cabeçalho na sua primeira linha onde é definido o formato de conteúdo e as informações referentes ao AIG, tudo separado por espaços, e todos os números inteiros trabalhados sejam positivos. O cabeçalho é iniciado por uma palavra que identifica o formato de conteúdo, sendo *aag* para ASCII e *aig* para binário. Em seguida são listados cinco números inteiros M, I, L, O e A, que podem ser interpretados da seguinte forma:

M - índice máximo de variáveis

I - número de entradas

L - número de *latches*

O - numero de saídas

A - número de operadores $\wedge$

Então, o cabeçalho de um arquivo AIGER é da seguinte forma:

```
formato M I L O A
```

Desse modo, o circuito possui I entradas, L *latches*, O saídas e constituído de A operadores $\wedge$. Se todas as variáveis forem utilizadas e não tiver operadores $\wedge$ sem uso, M pode ser definido como $M = I + L + A$ (BIERE, 2007). Bittencourt (2018) descreve que é

possível calcular o inteiro máximo que uma variável pode ter utilizando o índice máximo de variáveis (M) e este inteiro máximo é dado por $2 * M + 1$. As variáveis de entrada, *latch*, saída e operador não podem ser maiores que inteiro máximo.

Latch, em português trava, é uma porta com retroalimentação que possui duas entradas e uma saída e é comumente utilizado em circuitos sequenciais como dispositivo de memória. Em resumo, é um circuito sequencial biestável com a capacidade de armazenamento de informações dentro de um circuito. Com isso, um *latch* consegue assumir um de dois estados estáveis e que possui uma ou mais entradas que podem interferir na mudança do estado de saída (ROTH; KINNEY, 2014).

Conforme relatado por Biere (2007), com o formato AIGER é possível conseguir uma codificação de uma fórmula para o problema de SAT ao definir um circuito combinacional sem latches, isto é, $L = 0$, e com exatamente uma saída.

As variáveis são os índices presentes no intervalo de 1 até M. Deste modo, para obter um literal de uma variável é multiplicado o índice da variável por 2 e opcionalmente adicionar 1 se a variável for negada. E para extrair a variável de um literal é necessário dividí-lo por 2 (BIERE, 2007). Logo, pode-se concluir que números pares representam as variáveis e um número ímpar $x + 1$ representa a negação da variável x.

2.5.1 Formato em ASCII

Após o cabeçalho, as I linhas seguintes serão listados os literais de entrada não negados. Após os literais de entrada, são listadas L linhas referentes a lógica do latch e depois O linhas definindo os literais de saída. Após as O linhas serão listados A linhas referentes aos operadores $\wedge$ (BIERE, 2007).

É definido por Biere (2007) que o operador $\wedge$ binário é constituído por três números positivos, escritos um por linha e separados por exato um espaço. O primeiro inteiro é um número par e representa o literal ou lado esquerdo (LHS). O literal LHS é o literal de destino. Os outros dois inteiros representam os literais do lado direito (RHS) do operador $\wedge$, então conclui-se que os literais RHS são os literais de entrada.

O Código 2.1 demonstra como é a abstração da fórmula $\neg(\neg(p \wedge q) \wedge (\neg(q \wedge r)))$, apresentada inicialmente na Fórmula 2.2, em um arquivo AIGER no formato ASCII.

```
1       aag 6 3 0 1 3
2       2
3       4
4       6
5       13
6       8 2 4
7       10 4 6
8       12 9 11
```

Código 2.1 – Texto de um arquivo AIGER no formato ASCII referente à Fórmula 2.2. Fonte: Autor.

Portanto, conforme o código anterior, na primeira linha é descrito o cabeçalho do arquivo. Em seguida, da linha 2 à linha 4, são listados os literais de entrada que são 2, 4 e 6, onde na fórmula são as variáveis p, q e r, respectivamente. Na linha 13 é definido o literal de saída e da linha 6 à linha 8 são listados os operadores $\wedge$ da fórmula, onde o literal 8, na linha 6, é a conjunção das variáveis p e q, o literal 10, na linha 7, das variáveis q e r e o literal 12, na linha 8, a conjunção da negação de cada uma das conjunções anteriores. Estes operadores são representados pelos nós internos do DAG ilustrado na Figura 4, onde os literais 8, 10 e 12 são representados, respectivamente, pelos nós internos AND_8, AND_10 e AND_12.

2.5.2 Formato em binário

No formato em binário o arquivo é organizado de uma forma diferente do formato em ASCII, além de que é codificado de uma forma diferente da usual. Outra diferença é referente as variáveis e literais, no formato em ASCII é dado o índice das variáveis, já no formato em binário já é dado o literal. Na primeira linha ainda possui o cabeçalho, mas na linha seguinte invés das I linhas referentes a entrada, são listados L linhas referentes aos latches e O linhas referentes os literais de saída. Em seguida, em binário, são definidos os A operadores $\wedge$ (BIERE, 2007).

Biere (2007) assume que os operadores $\wedge$ estarão ordenados e respeitando a relação de pai-filho. Um operador $\wedge$ é constituído de três literais

$$lhs\ rsh0\ rsh1$$

sendo o literal lhs par e $lhs > rsh0 >= rsh1$. E como os índices lhs são todos consecutivos e inteiros pares, não há necessidade de manter o literal lhs no binário. Além disso, no binário são escritos $delta0$ e $delta1$ em vez dos literais $rsh0$ e $rsh1$ por conta da restrição de ordem. Assim, delta0 e delta1 são definidos da seguinte forma:

$$delta0 = lhs - rsh0,\ delta1 = rsh0 - rsh1.$$

Assim, entende-se que apenas os deltas são salvos na parte em binário e são codificados em um formato diferente, como mencionado anteriormente, mas segundo Bittencourt (2018), na biblioteca do AIGER são disponibilizados ferramentas que realizam a conversão de ASCII para binário e vice-versa. A Seção A.1 do Anexo A apresenta as funções que realizam a codificação de ASCII para binário e decodificação de binário para ASCII dos operadores $\wedge$.

2.5.3 Tabela de símbolos e Comentários

Após a definição dos operadores $\wedge$ existe uma tabela opcional para os símbolos. O símbolo é o rótulo de um nó e é definido por uma cadeia de caracteres do tipo ASCII. Símbolos podem ser definidos apenas para entradas, saídas e *latches* e é possível definir no máximo um símbolo por entrada, *latch* ou saída (BIERE, 2007).

A definição dos símbolos consiste em uma definição por linha onde é iniciado pelo tipo de símbolo i, l ou o, seguido por uma posição, sem espaço. A posição *pos* do símbolo é a posição da entrada, *latch*, ou saída, na lista de entradas, *latches*, e saídas respectivamente. Após a posição, é adicionado um espaço e, por fim, uma cadeia de caracteres que é o rótulo. Portando, uma definição de símbolo é da seguinte forma (BIERE, 2007):

```
[ilo]<pos> <string>
```

Da mesma maneira que a tabela de símbolos, os comentários são opcionais. Para adicionar comentários ao arquivo é adicionado um caractere c seguido por uma quebra de linha, assim todas as linhas seguintes serão comentários. O último comentário tem de ser encerrado por um caractere de quebra de linha, onde este caractere deve ser o último caractere do arquivo (BIERE, 2007).

Segundo Biere (2007), não há distinção da tabela de símbolos e comentários no formato ASCII e binário. Assim, refatorando o Código 2.2, o código com a tabela de símbolos e comentários ficaria da seguinte forma:

```
1       aag 6 3 0 1 3
2       2
3       4
4       6
5       13
6       8 2 4
7       10 4 6
8       12 9 11
9       i0 VARIAVEL_p
10      i1 VARIAVEL_q
11      i2 VARIAVEL_r
12      o0 out
13      c
14      Código referente a fórmula ¬(¬(p ∧ q) ∧ (¬(q ∧ r)))
```

Código 2.2 – Texto de um arquivo AIGER no formato ASCII referente a fórmula 2.2 com a tabela de símbolos e comentários. Fonte: Autor.

Assim, conforme o código anterior, da linha 9 à linha 11 é listado a tabela de símbolos, em que a partir da linha 9 à linha 10 rotulam, respectivamente, os literais de entrada 2, 4 e 8 e na linha 12 é rotulado o literal de saída 13. Na linha 13 é iniciado a seção de comentários e na linha seguinte é descrito um breve comentário.

2.6 Considerações

Ao longo deste capítulo foi apresentado o embasamento teórico necessário para a implementação da proposta deste trabalho.

Primeiramente é explicado sobre a Lógica Proposicional como um ramo da lógica formal onde se estuda as chamadas proposições, ou fórmulas. Uma fórmula é constituída de variáveis, onde podem receber uma valoração verdadeira ou falsa, e operadores lógicos, por exemplo, $\wedge$, $\vee$, $\neg$, $\Rightarrow$ e $\Leftrightarrow$, para criar fórmulas mais complexas. As fórmulas podem ser representadas através de grafos dirigidos acíclicos e existem diferentes tipos de representação para isto. O primeiro tipo e mais utilizado é a Forma Normal Conjuntiva (CNF) também chamada forma clausal, que consiste na conjunção de disjunções. Existem fórmulas que não estão na representação CNF, estas são chamadas fórmulas não-clausais e podem ser representadas pelas representações não-clausais ISCAS, EDIMACS e AIG.

Em seguida é apresentado o Problema da Satisfatibilidade Booleana (SAT) onde é um problema NP-completo que consiste em identificar se uma fórmula é satisfatível ou não mediante a valoração aplicada a suas variáveis. Existe formas automatizadas para verificar

se uma fórmula é satisfatível e isto é feito através de resolvedores de satisfatibilidade. Os resolvedores têm diferentes abordagens para determinar a satisfatibilidade de uma fórmula, assim como os resolvedores não-clausais tem a possibilidade de agir diretamente na fórmula ou realizar a conversão para a representação CNF. É uma área que tem avançado bastante, principalmente por conta de competições que incentivam a criação de novos resolvedores e algoritmos para a resolução do problema de satisfatibilidade.

E, por fim, é apresentado o formato AIGER, onde é um formato de arquivo e biblioteca voltado para representar e auxiliar na manipulação de fórmulas na representação AIG. O formato AIGER pode ser utilizado tanto para a lógica combinacional quanto para a sequencial devido à possibilidade do uso de *latches* na composição da fórmula. É um formato que possui dois tipos de representações no arquivo, binário e ASCII, onde a forma binária é criada por uma codificação própria.

Entretanto, é necessário um melhor entendimento do funcionamento de um resolvedor não-clausal e para isto é demonstrado no próximo capítulo como é a lógica e o processo de resolução do problema de satisfatibilidade aplicado a uma fórmula não-clausal.

3 Resolvedor não-clausal LIAMFSAT

Este capítulo visa compreender o funcionamento de um resolvedor não-clausal. São apresentados informações sobre a sua estrutura lógica, técnicas utilizadas para o processo de resolução de uma fórmula e um exemplo para esclarecer o funcionamento do resolvedor. Portando, o resolvedor não-clausal escolhido para isto foi o LIAMFSAT[1]. As explicações referentes ao LIAMFSAT tem como referência a dissertação *Satisfatibilidade não-clausal restrita às variáveis de entrada* de Bruno César Ribas (RIBAS, 2011).

3.1 Sobre o LIAMFSAT

O LIAMFSAT é um resolvedor não-clausal onde seu nome é a união das siglas do Laboratório de Inteligência Artificial e Métodos Formais (LIAMF) da Universidade Federal do Paraná (UFPR) com Satisfatibilidade (SAT).

Este resolvedor tem a abordagem de atuar diretamente na fórmula original, onde implementa diversas técnicas de resolvedores modernos, por exemplo: retrocesso não-cronológico; aprendizado de cláusula; BCP rápido; e escolha de variável bem definida. O LIAMFSAT utiliza como formato o ISCAS devido a sua grande facilidade em representar qualquer fórmula lógica.

É necessário uma estrutura da fórmula ser representada em memória para que seja possível que o resolvedor procure por uma valoração para as variáveis da fórmula que torne o nó raiz da estrutura verdadeiro. Então, após a leitura de um arquivo contendo a fórmula em ISCAS, é gerado um grafo dirigido acíclico (DAG) onde cada nó do grafo que representa a fórmula contém os seguintes atributos:

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

[1] Código do LIAMFSAT disponível em: https://github.com/UnB-SAT/liamfsat.

3.2 Processo de busca

O processo de busca do LIAMFSAT ocorre de forma semelhante ao utilizado por resolvedores baseados no DPLL. Diferente de algoritmos baseados em fórmulas CNF, onde todos os operadores conectados ao nó raiz devem possuir uma valoração verdadeira para que o nó raiz também tenha valoração verdadeira, em fórmulas não-clausais pode ser necessário que algum operador seja valorado como falso para que o nó raiz, seja valorado como verdadeiro. O LIAMFSAT possui a restrição de valorar apenas as variáveis, isto se deve a dificuldade de se saber qual valor ideal para os nós internos, mas utiliza do campo de valoração presente em cada nó para propagar o efeito da valoração da variável.

Uma etapa anterior ao processo de busca na fórmula é realizado, onde o nó raiz e uma variável qualquer são valorados como verdadeiros. Deste modo, é realizada a análise do efeito dessa valoração e o processo continua até que seja encontrada uma valoração que torne a fórmula verdadeira ou até que ocorra um conflito.

Pelo fato de possuir a restrição de decidir a valoração apenas das variáveis, é garantido que a propagação ocorra em apenas uma direção, ou seja, a propagação ocorre apenas de filho para pai e nunca ao contrário. Assim, é possível que cada nó acumule todas as modificações indicadas por seus nós filhos.

O LIAMFSAT utiliza da ideia de visitar cada nó apenas uma vez para executar os sinais pendentes, onde estes sinais são utilizados para a propagação das modificações no grafo. Para ser realizada apenas uma visita, o DAG deve ser reordenado a partir de uma ordem de verificação, deste modo, será possível uma verificação em ordem crescente até chegar na raiz do grafo. Todas as variáveis são marcadas como nível 0 e o nível dos nós internos é definido pela fórmula:

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

A Figura 7 é um diagrama nivelado baseado na Fórmula 2.1, dada pela fórmula $((p \vee q) \wedge (r \vee \neg s) \wedge \neg q) \vee \neg(\neg p \wedge (r \vee \neg s) \wedge q)$, e em seu diagrama, dado na Figura 2.

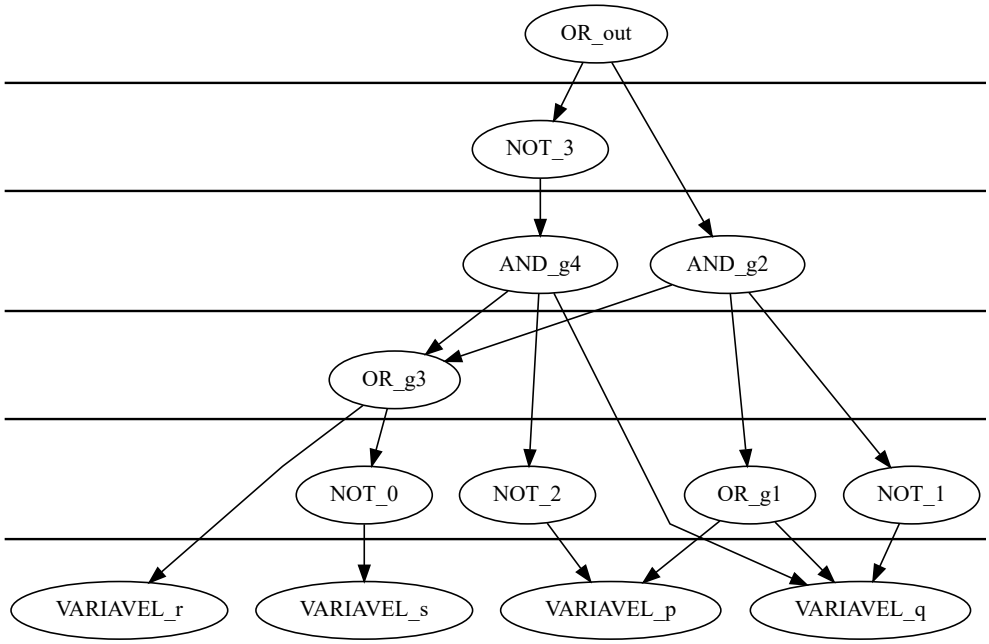

Figura 7: Diagrama da fórmula $((p \vee q) \wedge (r \vee \neg s) \wedge \neg q) \vee \neg(\neg p \wedge (r \vee \neg s) \wedge q)$. Fonte: Adaptado de (RIBAS, 2011).

Além da ordem de verificação, o grafo possui uma estrutura de fila por nível para que cada nó que possua sinais pendentes sejam colocados, assim garantindo uma única verificação por nó.

Um sinal é enviado para todos os nós pais sempre que um nó mudar a valoração de desconhecido para um valor verdade, seja ele verdadeiro ou falso. Cada operador deve possuir uma estrutura para guardar os sinais pendentes. O processo de buscar em cada nível pelos operadores com sinais pendentes e verificar se os sinais forçam a propagação de valoração é realizado pelo *Boolean Constraint Propagation* (BCP) e continua até que não existam mais sinais pendentes.

Como o LIAMFSAT está restrito a valoração apenas das variáveis, isto significa que o motivo da valoração das variáveis será sempre por decisão, diferente da causa da valoração de um operador que será sempre por inferência por levar em consideração a valoração de todos os nós filhos. Logo, o momento crítico no estado de valoração ocorre apenas quando o nó raiz se tornar falso, assim sendo necessário a realização de um retrocesso, que consiste em tentar outra valoração onde ocorreu uma contradição. A ordem de decisão das variáveis é feita de forma aleatória iniciando por verdadeiro e depois por falso. É definido que o nível de decisão será dado pelo operador e pelo valor associado:

($\wedge$, **verdadeiro**): nível de decisão de todos os filhos;

($\wedge$, **falso**) : nível de decisão do filho falso de menor nível de decisão;

($\vee$, **verdadeiro**) : nível de decisão do filho verdadeiro de menor nível de decisão;

($\vee$, **falso**) : nível de decisão de todos os filhos.

A análise de conflito consiste em identificar o erro cometido ao valorar alguma variável. Primeiramente é verificado se a última decisão já foi testada para os dois valores verdade, isto é, verdadeiro e falso. Caso apenas um valor foi testado, o algoritmo troca o valor da última decisão e o processo de busca continua. Se por acaso já tenha sido testados ambos valores, então o algoritmo procura pelo maior nível de decisão entre os filhos que ajudaram com a valoração falsa da raiz.

Ao identificar a causa da valoração conflitante, é realizado um processo de retrocesso até o nível que ocasionou o conflito e alterando o valor de verdade do nó. Pode acontecer do retrocesso chegar no primeiro nível de decisão, onde esta decisão já foi testada com ambos valores verdade, neste caso a fórmula é instatisfatível e o algoritmo é encerrado.

O LIAMFSAT possui também, além da capacidade de analisar e identificar o nível de decisão onde se deve efetuar o retrocesso, a capacidade de aprender uma cláusula objetivando que o mesmo erro não seja cometido diversas vezes. No momento que ocorre um conflito, a busca percorre o DAG até que o nível de decisão dos nós se torne diferente do nível atual, assim, as variáveis valoradas nesses níveis definem a cláusula que representa o conflito e então é adicionada a fórmula.

Desta maneira, o LIAMFSAT define então se uma fórmula é satisfatível através de um processo de análise de valoração das variáveis e a propagação até o nó raiz, caso não seja encontrado uma valoração que torne verdadeiro a valoração do nó raiz, é retornado que a fórmula é insatisfatível.

A notação utilizada para representar o valor de verdade de uma variável e o nível de decisão em que foi valorado é dada pelo valor de verdade e o nível de decisão em que foi valorado separados pelo símbolo @. Tomemos como exemplo a valoração $p_{v@2}$, é definido o valor verdadeiro à variável p e a valoração aconteceu no nível de decisão 2 (MARQUES-SILVA; LYNCE; MALIK, 2009).

Assim, para elucidar o processo de busca, retrocesso não-cronológico e aprendizado de cláusulas do LIAMFSAT e baseado no exemplo dado por Ribas (2011), tomemos como exemplo a Fórmula 2.1, onde foi representada na Figura 7 de forma nivelada. Dadas as seguintes valorações parciais para esta fórmula:

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

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

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

Esta valoração gerou um conflito, como pode ser ilustrado na Figura 8, onde são apresentadas as valorações anteriores e sua propagação até o nó raiz.

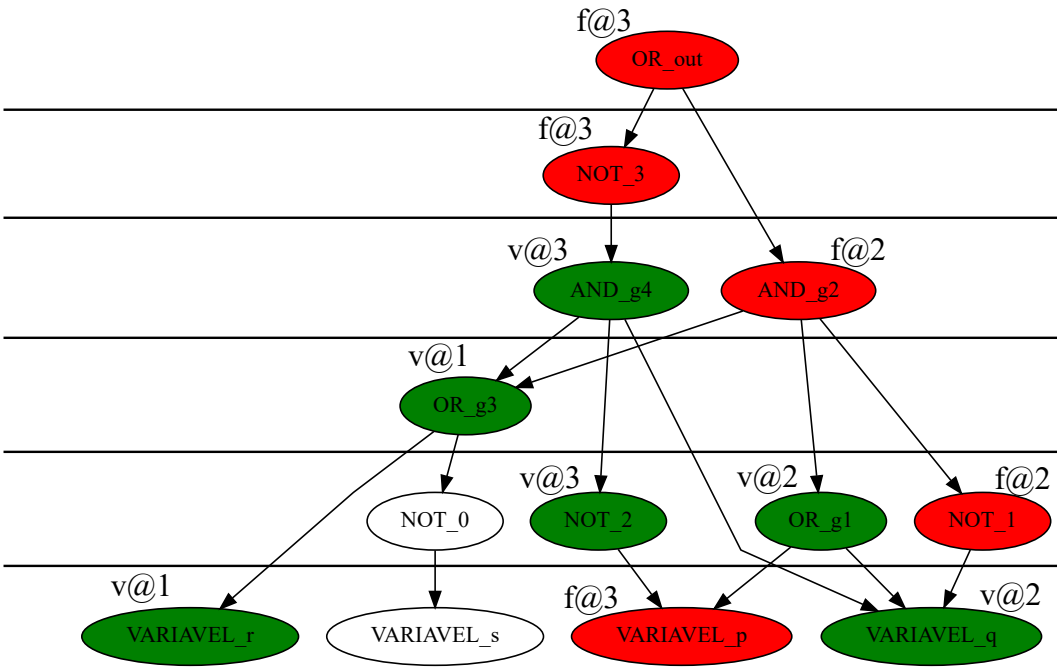

Figura 8: Diagrama nivelado com as valorações $r_{v@1}$, $q_{v@2}$ e $p_{f@3}$. Fonte: Adaptado de (RIBAS, 2011).

Para ilustrar o diagrama anterior, cada nó valorado é representado por uma cor a ser determinada pelo seu valor de verdade, verde para verdadeiro e vermelho para falso. Ao lado de cada nó valorado é adicionado um texto referente ao seu valor verdade e o nível de decisão em que foi valorado.

Ao analisar a propagação das valorações em cada nível de decisão na Figura 8, o conflito foi criado no nível de decisão 3. Desta forma, o resolvedor começa a busca pela causa do conflito descendo a partir do nós raiz a procura dos níveis de decisão que influenciaram a valoração da raiz. Ao percorrer o grafo até as variáveis e analisar o nível de decisão de cada uma delas, é constatado que o conflito foi ocasionado pela valoração falsa da variável p no nível de decisão 3. Caso a variável p não tenha sido testada com os dois valores, verdadeiro e falso, é alterado o seu valor. Para este exemplo, suponhamos que a variável tenha sido testada apenas para falso, logo, sua valoração passa a ser verdadeira.

Durante o processo de percorrer o grafo, são adicionadas à cláusula aprendida as variáveis identificadas como participantes do conflito, neste exemplo a cláusula $(\neg r \vee \neg q \vee p)$ foi gerada. A cláusula aprendida é adicionada na raiz da fórmula, mas caso a raiz seja um operador $\vee$ é criado um operador $\wedge$ para ser a raiz da fórmula e nesta nova raiz são adicionadas a raiz anterior e todas todas as cláusulas aprendidas.

Com o processo de busca, retrocesso não-cronológico e aprendizagem de cláusula,

foi gerado o diagrama ilustrado na Figura 9.

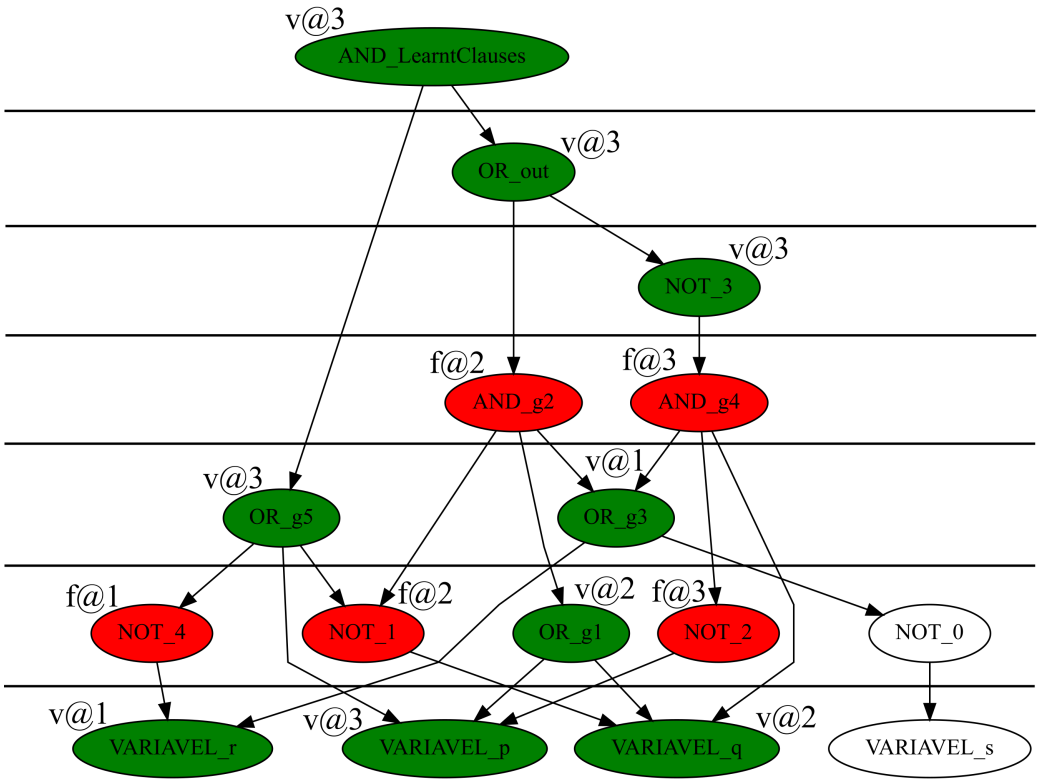

Figura 9: Diagrama nivelado com as valorações $r_{v@1}$, $q_{v@2}$ e $p_{v@3}$. Fonte: Adaptado de (RIBAS, 2011).

Analisando a Figura 9 é visto que foi criado um nó raiz novo, AND_LearntClauses, devido o nó raiz original ser um operador $\vee$. O nó raiz criado é um operador $\wedge$ que tem como filhos o nó raiz antigo, OR_out, e a cláusula aprendida, OR_g5.

3.3 Considerações

Foi abordado neste capítulo um resolvedor não-clausal para esclarecer o processo de resolução de fórmulas não-clausais. O resolvedor escolhido foi o LIAMFSAT, onde é um resolvedor com um processo de resolução semelhante ao utilizado por resolvedores baseados no algoritmo Davis-Putnam-Logemann-Loveland (DPLL). É um resolvedor que possui a abordagem de atuar diretamente na fórmula, descartando a conversão para representação CNF, além da utilização de diversas técnicas presentes em resolvedores modernos, como o retrocesso não-cronológico, aprendizado de cláusula e BCP rápido. O LIAMFSAT possui a restrição de valorar apenas as variáveis, isto é devido à dificuldade em saber a valoração ideal dos nós internos, assim, a forma de propagação acontece sempre de filho para pai e nunca ao contrário.

4 Alterações no LIAMFSAT

Este capítulo é destinado a descrever todas as alterações e melhorias realizadas no resolvedor não-clausal LIAMFSAT, se baseando no referencial teórico apresentado no Capítulo 2 e sem modificar a estrutura e técnicas apresentadas no Capítulo 3. Na Seção A.1 do Apêndice A é descrito o processo de compilação e execução do resolvedor.

4.1 Restauração de código antigo

A última alteração feita foi no ano de 2010 e por se tratar de um código com mais de 10 anos foi necessário realizar uma refatoração no código e na organização dos arquivos.

Anteriormente, todos os arquivos do resolvedor ficavam na pasta raiz e este formato de organização de arquivos pode ser um problema considerando o crescimento do resolvedor e adição de novas funcionalidades. Foi realizado uma organização dos arquivos em pastas seguindo a função de cada arquivo. Essas alterações foram realizadas de modo a facilitar trabalhos futuros no LIAMFSAT. As pastas criadas foram as seguintes:

- **inc** - arquivos de cabeçalho;
- **src** - arquivos de código fonte;
- **Makefile** - arquivos com instruções de compilação;
- **libs** - bibliotecas;
- **tools** - ferramentas auxiliares.

A criação da pasta *libs* foi uma forma de facilitar a adição de novas funcionalidades no formato de biblioteca. Até antes do desenvolvimento deste trabalho, a única biblioteca presente no resolvedor é o conversor de ISCAS. Já a pasta *tools*, foi criada com o intuito de adicionar ferramentas auxiliares ao LIAMFSAT. Algumas ferramentas auxiliares criadas neste trabalho estão descritas na Seção 4.4.

4.2 Empacotamento dos conversores

Foi criado um empacotamento dos conversores das representações de entrada e esta implementação facilitou a adição de novos tipos de entrada.

Para adicionar um novo tipo de entrada basta implementar o conversor no formato de biblioteca e adicionar na pasta *libs*. Após a implementação, basta adicionar uma nova condicional no empacotador para ser chamado o conversor referente ao tipo de entrada desejado.

O arquivo que realiza o empacotamento é o *parser.c* que está presente na pasta *src*. O Código 4.1 descreve a condicional que realiza a chamada do conversor correto para o tipo de entrada presente no empacotador, onde na linha 3 é realizado a verificação da extensão do arquivo e na linha 4 é chamado o conversor de ISCAS caso seja do tipo ISCAS. É necessário informar o caminho até o arquivo de entrada na variável *filename*.

```
1 char* file_extension = strrchr(filename, '.');
2 if (file_extension != NULL) {
3     if (!strcmp(file_extension, ".iscas")) {
4         formula = iscas_parser(filename);
5     } else {
6         printf("This extension is not supported: %s\n", file_extension);
7         exit(1);
8     }
9 } else {
10    printf("This extension is not supported: %s\n", file_extension != NULL ?
      file_extension : "");
11    exit(1);
12 }
```

Código 4.1 – Condicional no empacotador para chamar o conversor de ISCAS. Fonte: Autor.

4.3 Representação AIG como parâmetro de entrada

Diante da escolha da representação AIG (Seção 2.3.3 deste trabalho) foi necessário implementar um novo conversor para este novo formato de entrada.

Foi optado pelo formato AIGER (Seção 2.5 deste trabalho) como o tipo de arquivo a ser alimentado no resolvedor. Este formato foi escolhido devido a sua facilidade de manipulação e a disponibilização de uma biblioteca que auxiliou na implementação desta nova representação. Outro ponto a ser mencionado é a variedade de arquivos de testes que poderão ser usados nos experimentos, alguns utilizados inclusive em competições de satisfatibilidade, por exemplo, no SAT-Race do ano de 2010 (SINZ, 2010). Além disso, o Instituto de Modelos Formais e Verificação da Universidade de Linz, em alemão Johannes Kepler Universität Linz ou JKU, disponibiliza um conversor do formato CNF para o formato AIG em que este pode ser utilizado para converter em AIG fórmulas que eram testadas em CNF até então, assim aumentando os arquivos de testes disponíveis (CNF2AIG, 2008).

Além do formato AIG foram analisados outros dois formatos de entrada para o resolvedor, o ISCAS e o EDIMACS (respectivamente, Seções 2.3.1 e 2.3.2 deste trabalho).

Os dois formatos possuem os mesmos problemas: falta de uma boa documentação ou estudos sobre o formato e pouca quantidade de arquivos de testes para os experimentos. São formatos que possuem fácil compreensão, mas com a invenção da representação AIG e seus estudos em diversos trabalhos tornou-se uma ótima opção como tipo de entrada do resolvedor.

Assim como o conversor de ISCAS, o conversor de AIG para a estrutura de grafo do LIAMFSAT foi adicionado no formato de biblioteca no LIAMFSAT sendo adicionado também uma nova condicional no empacotador para chamar o conversor. Desta forma, o Código 4.2 descreve como ficou a condicional após a implementação do novo conversor, onde na linha 5 é realizado a verificação da extensão do arquivo e na linha 6 é chamado o conversor de AIG caso seja do tipo AIG.

```
1 char* file_extension = strrchr(filename, '.');
2 if (file_extension != NULL) {
3     if (!strcmp(file_extension, ".iscas")) {
4         formula = iscas_parser(filename);
5     } else if (!strcmp(file_extension, ".aig") || !strcmp(file_extension, ".aag")) {
6         formula = aig_parser(filename);
7     } else {
8         printf("This extension is not supported: %s\n", file_extension);
9         exit(1);
10    }
11 } else {
12    printf("This extension is not supported: %s\n", file_extension != NULL ?
      file_extension : "");
13    exit(1);
14 }
```

Código 4.2 – Condicional no empacotador para chamar o conversor correto. Fonte: Autor.

4.3.1 Dificuldades encontradas

Por se tratar de um resolvedor voltado para problemas de satisfatibilidade e a escolha do formato AIGER como formato de entrada, surgem duas dificuldades: a existência de *latches* e múltiplas saídas no arquivo.

Para contornar a dificuldade relacionada a existência de *latches* na fórmula, o resolvedor proposto será delimitado a fórmulas que não possuem *latches* em sua composição. Como explicado por Roth e Kinney (2014), a existência de *latches* torna a lógica da fórmula como sequencial onde a saída não depende exclusivamente das combinações das variáveis de entrada como acontece na lógica combinacional. Assim, o cabeçalho do arquivo de entrada deve possuir $L = 0$.

Em casos que o arquivo tenha mais de uma saída, todas as saídas são ligadas a um operador $\vee$, onde este operador se torna o nó raiz do grafo. Desta forma, será possível saber se a fórmula é satisfatível ou não verificando apenas a valoração do nó raiz, descartando

a necessidade de verificar todas as saídas existentes no arquivo.

4.3.2 Geração do grafo

A geração do grafo inicia-se com a leitura da estrutura AIG. A biblioteca AIGER disponibiliza que a leitura possa ser realizada através de uma cadeia de caracteres ou pela leitura de um arquivo, onde foi optado pela leitura do arquivo para o resolvedor proposto. Esta leitura é realizada com o auxílio da função de decodificação presente na biblioteca onde o código está detalhado na função *decode* na Seção A.1 do Anexo A.

Após a leitura do arquivo, é retornado um registro do tipo *aiger*, onde a definição deste registro está especificada no Código 4.3.

```
1  struct aiger_and {
2    unsigned lhs;
3    unsigned rhs0;
4    unsigned rhs1;
5  };
6
7  struct aiger_symbol {
8    unsigned lit;
9    unsigned next;     /* apenas utilizado em latches */
10   char *name;
11 };
12
13 struct aiger {
14   unsigned maxvar;
15   unsigned num_inputs;
16   unsigned num_latches;
17   unsigned num_outputs;
18   unsigned num_ands;
19
20   aiger_symbol *inputs;
21   aiger_symbol *latches;
22   aiger_symbol *outputs;
23   aiger_and *ands;
24
25   char **comments;
26 };
```

Código 4.3 – Registro retornado após leitura de arquivo AIGER. Fonte: Biere (2007).

Com o registro retornada da leitura do arquivo é realizada uma conversão para a *struct* de grafo do LIAMFSAT (Seção 3 deste trabalho). Assim, com a representação da fórmula em memória, é possível o resolvedor procurar por uma valoração parcial para as variáveis que torne o nó raiz da estrutura verdadeiro.

O algoritmo de conversão possui um laço de repetição que interage com a lista de operadores $\wedge$ da fórmula, onde cada operador possui o seu próprio lhs, $rhs0$ e $rhs1$,

[1] Licença do código fonte da biblioteca AIGER está disponível em: https://github.com/arminbiere/aiger/blob/master/LICENSE.

[2] Código fonte completo da biblioteca AIGER está disponível em: https://github.com/arminbiere/aiger.

utilizados para criar os nós e arestas do grafo. Como explicado na Seção 2.5.1, lhs, $rhs0$ e $rhs1$ são literais que compõem um operador $\wedge$, sendo lhs o literal de destino e os outros dois, $rhs0$ e $rhs1$, os literais de entrada do operador. O operador $\wedge$ possui uma *struct* própria e é definida da linha 1 a 5 no código 4.3.

Para cada lhs é criado um nó sendo vinculado a este nó todos os nós filhos, isto é, $rhs0$ e $rhs1$. Caso exista uma relação complementada, ou seja, quando ocorre inversão de valor, é criado um nó para representar esta inversão. O algoritmo inicial da função que converte para grafo está especificado no Algoritmo 3.

```
   Entrada: Fórmula em AIGER
   Saída: DAG inicializado
 1 para cada operador ∧ faça
 2 |   no ← CriarNo(lhs, Operador_AND)
 3 |   se rhs0 é par então
 4 |   |   AdicionarFilho(no, rhs0)
 5 |   senão
 6 |   |   nó_inversor ← CriarNo(rhs0, Operador_NOT)
 7 |   |   AdicionarFilho(no_inversor, rhs0 − 1)
 8 |   |   AdicionarFilho(no, no_inversor)
 9 |   fim
10 |   se rhs1 é par então
11 |   |   AdicionarFilho(no, rhs1)
12 |   senão
13 |   |   no_inversor ← CriarNo(rhs1, Operador_NOT)
14 |   |   AdicionarFilho(no_inversor, rhs1 − 1)
15 |   |   AdicionarFilho(no, no_inversor)
16 |   fim
17 fim
18 se literal de saída é ímpar então
19 |   AdicionarFilho(literal de saída, literal de saída - 1)
20 fim
21 retorna DAG
```

Algoritmo 3: Função de conversão para DAG

Inicialmente, o algoritmo de conversão possui duas funções específicas, *AdicionarFilho* e *CriarNo*, onde estas têm a função de criar os nós e o nó inversor, caso a aresta seja complementada, no grafo e adicionar os nós filhos a um determinado nó.

A função *AdicionarFilho* adiciona o nó filho a um nó específico e tem por parâmetros o nó que será adicionado o filho e o filho. É aproveitado o momento onde o filho é adicionado para criar a relação de filho para pai, isto é, adicionar o pai a lista de pais no

nó filho. Na função *CriarNo* é criado o nó caso não exista no grafo e retorna-o, mas caso o nó já exista, apenas é retornado-o. Esta função tem por parâmetros o literal e o tipo do nó.

4.4 Ferramentas auxiliares

Durante o desenvolvimento deste trabalho houve-se a necessidade de testar a mesma fórmula em diferentes representações e, para isto, foram desenvolvidas duas ferramentas que convertem uma representação para outra e adicionado uma ferramenta pertencente a biblioteca AIGER. As ferramentas são: conversores de ISCAS para AIG, de AIG para ISCAS e de binário para ASCII ou de ASCII para binário voltada para a representação AIG.

Estas ferramentas foram implementadas visando comparar o tempo gasto para converter a representação de entrada para a estrutura de grafo e verificar se existe diferença entre os conversores em questão de tempo.

A ferramenta que converte de ISCAS para AIG possui uma limitação de aceitar apenas fórmulas simples, onde um operador $\wedge$ ou $\vee$ tenham no máximo dois filhos. Uma fórmula pode ser escrita de diferentes formas na representação ISCAS por ser menos restritiva, algo que a representação AIG não é. Desta forma, uma conversão de uma fórmula em ISCAS com operadores que possuem diversos filhos poderia resultar em uma fórmula de maior tamanho devido à quantidade de operadores necessários para representar este operador mesmo operador na representação AIG.

4.5 Funcionalidade de propagação

Foi implementado uma nova funcionalidade que exporta a propagação realizada pelo BCP em cada etapa de valoração parcial ou de retrocesso. O objetivo desta funcionalidade é visualizar se a propagação pela fórmula no processo de resolução está sendo feita conforme o esperado, além da possibilidade de ser utilizada para explicar o processo de propagação em uma fórmula. A seguir é listado o processo de propagação e resolução da fórmula

$$(\neg a \wedge b) \vee (\neg b \wedge \neg c)$$

a iniciar pela Figura 10, onde os nós de cor verde e vermelho representam, respectivamente, o valor de verdade $verdadeiro$ e $falso$, e o nó com borda tracejada representa a variável que está sendo valorada naquele momento.

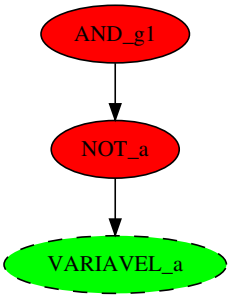

Figura 10: Valoração parcial da variável a. Fonte: Autor.

Na Figura 11 é realizada a valoração parcial da variável b e com isso a propagação torna o nó de saída com o valor de verdade $falso$.

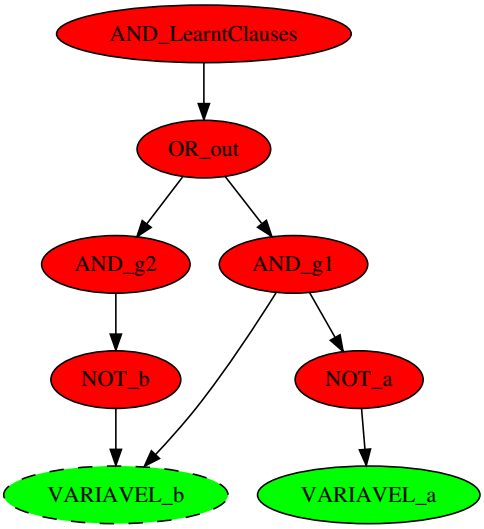

Figura 11: Valoração parcial da variável b. Fonte: Autor.

É então realizado o retrocesso para realizar uma nova valoração parcial para variável b com o valor de verdade restante, no caso o valor $falso$. A Figura 12 é após o processo de retrocesso.

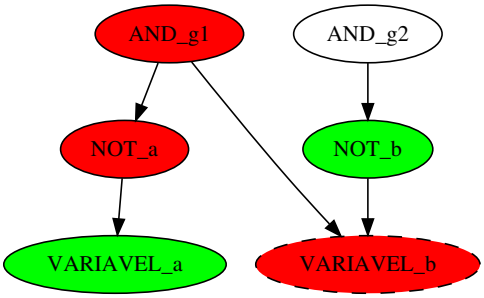

Figura 12: Valoração parcial da variável b com $falso$ após retrocesso. Fonte: Autor.

O resolvedor então segue para a próxima variável, no caso a variável c, onde está representada na Figura 13.

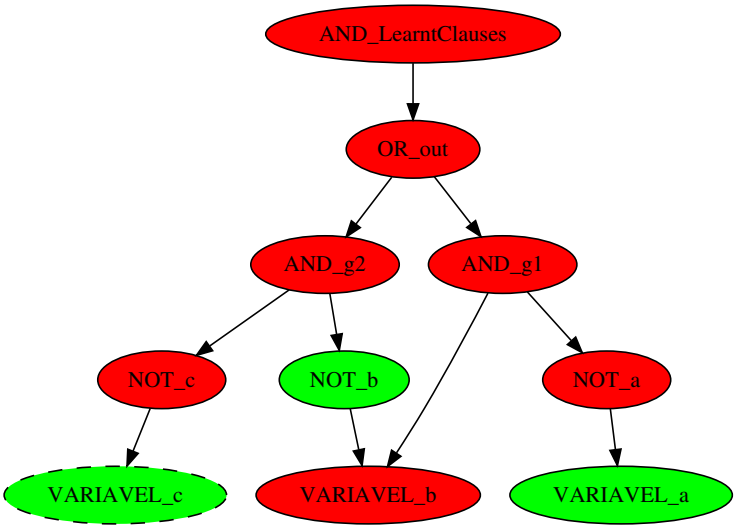

Figura 13: Valoração parcial da variável c. Fonte: Autor.

É realizado um novo retrocesso para testar a variável c com o valor de verdade $falso$. Com esta valoração parcial, a fórmula se tornou satisfatível, como é apresentado na Figura 14, e assim o processo é encerrado.

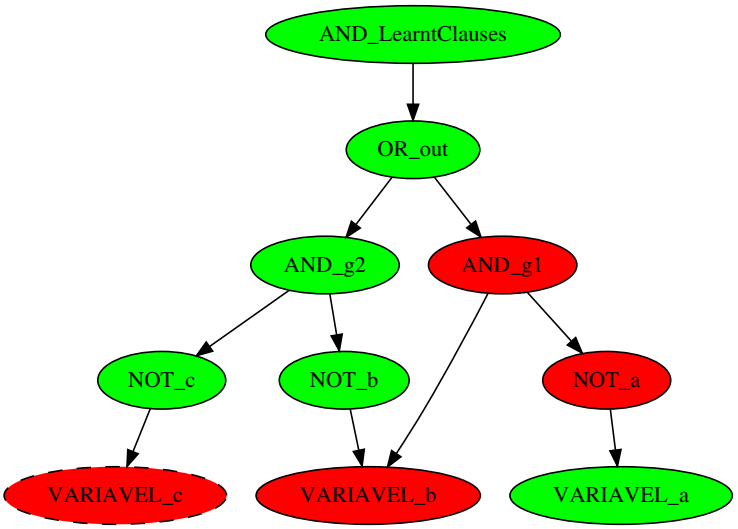

Figura 14: Valoração parcial da variável c com $falso$ após retrocesso. Fonte: Autor.

Estas figuras foram geradas com o auxílio da funcionalidade de propagação e com ela foi possível encontrar um erro no processo de retrocesso que ao retroceder para a primeira decisão realizada, o resolvedor encerra o processo de resolução sem tentar o valor de verdade restante.

5 Experimentos

Foram realizados experimentos visando calcular o tempo gasto e uso de memória pelo resolvedor LIAMFSAT no processo de resolução de um conjunto de problemas.

Com finalidade de executar os experimentos foi utilizado um computador com processador Intel Core i7-8700 3.2GHz e 16 GB de memória RAM. Para este tipo de experimento foi utilizado um tempo limite para cada execução de 30 minutos.

Foram realizados um total de nove testes onde os arquivos utilizados fazem parte do conjunto de problemas utilizados na competição de satisfatibilidade SAT-Race no ano de 2010 (2010).

A Tabela 1 apresenta o conjunto de problemas testados juntamente com a quantidade de entradas e operadores $\wedge$ na composição da fórmula.

Problema	**Entradas**	**Operadores $\wedge$**
a216test0002.aig	27	629
a452test0016.aig	31	5918
par8-5-c.sat_opt.aig	150	4165
AProVE07-27.aig	2825	37787
neclaftp3002.aig	12275	328323
cube-11-h14-sat.aig	1376	982579
bgpd_bgpd_vc75772.aig	192	38240
countbits512.aig	512	1562177
countbits1024.aig	1024	6270017

Tabela 1 – Conjunto de problemas testados no LIAMFSAT. Fonte: Autor.

A Tabela 2 descreve o resultado de cada problema, assim como tempo gasto em segundos e o consumo de memória do LIAMFSAT durante a execução.

Problema	**Resultado**	**Tempo**	**Uso de memória**
a216test0002.aig	SAT	0,01	3,67 MB
a452test0016.aig	SAT	0,01	4,74 MB
par8-5-c.sat_opt.aig	UNSAT	0,01	13,44 MB
AProVE07-27.aig	UNSAT	0,21	113,98 MB
neclaftp3002.aig	SAT	1,29	1,97 GB
cube-11-h14-sat.aig	Tempo esgotado	-	-
bgpd_bgpd_vc75772.aig	Erro	0,05	81,62 MB
countbits512.aig	Erro	1,91	3,1 GB
countbits1024.aig	Erro	6,89	12,3 GB

Tabela 2 – Resultado do teste de desempenho no LIAMFSAT. Fonte: Autor.

Outros problemas foram testados além dos listados na Tabela 1, mas houve uma grande quantidade de problemas que retornaram erro, assim como os problemas *bgpd_bgpd_vc*75772.*aig* e *countbits*512.*aig*. Este erro está relacionado com a finalização do algoritmo DPLL sem o nó de saída ter sido valorada no resolvedor e uma possível causa para este erro pode ser o erro identificado na Seção 4.5.

Com estes testes foi possível medir também o uso de memória utilizado pelo resolvedor e para alguns problemas foi identificado um alto consumo de memória, como, por exemplo o problema *neclaftp*3002.*aig* que teve um pico de consumo de 1,97 GB. Os problemas *countbits*512.*aig* e *countbits*1024.*aig* mesmo retornando erro houve um pico de consumo de 3,1 GB e 12,3 GB durante a sua execução, respectivamente.

É proposto por Youness et al. (2020) um resolvedor paralelo onde o pré-processamento é realizado de forma paralela utilizando sistemas heterogêneos de CPU e GPU no processo de resolução do problema de satisfatibilidade. Este resolvedor obteve um aumento de desempenho de até 15x se comparado a implementação sequencial utilizando esta abordagem. Em trabalhos futuro é possível a realização de estudos de modo a verificar se esta abordagem é aplicável ao LIAMFSAT com o objetivo de melhorar seu desempenho e uso de memória.

6 Considerações Finais

Neste trabalho foi apresentado o Problema da Satisfatibilidade Booleana (SAT) e a sua importância para diferentes áreas, além dos diversos estudos para aperfeiçoar a resolução deste problema. Este problema encontra uma valoração para as variáveis de uma fórmula que a torne satisfatível.

Diante dos estudos realizados, foi visto que existem ferramentas que resolvem o problema de satisfatibilidade de forma automática e para isto foi escolhido o resolvedor não-clausal LIAMFSAT como objeto de estudo para ser implementado um novo formato de entrada, o formato AIG.

O objetivo principal deste trabalho, a implementação do formato AIG no resolvedor LIAMFSAT, foi realizado de forma completa e com esta implementação foi possível encontrar alguns problemas no resolvedor. Desta forma, a parte do código do algoritmo DPLL presente no resolvedor deverá ser refatorada para incluir técnicas mais novas. Além de que o código agora está mais legível e pronto para receber melhorias.

Com o formato AIG implementado de forma completa foi possível realizar experimentos que permitiram chegar a conclusão de que o LIAMFSAT possui a necessidade de uma otimização no consumo de memória e a correção do processo de retrocesso, identificado através da nova funcionalidade de propagação.

Com os resultados obtidos por experimentos foi possível identificar melhorias que podem ser realizadas em trabalhos futuros.

6.1 Trabalhos futuro

Com este trabalho foi possível identificar um erro e uma melhoria a ser realizada no resolvedor LIAMSAT em possíveis trabalhos futuros. O erro identificado está relacionado com o processo de retrocesso que não é testado o valor de verdade restante ao retroceder a primeira decisão e assim encerrando o processo de resolução precocemente.

Como apresentado na Seção 5, após realizar alguns experimentos de desempenho para verificar o tempo de execução e o uso de memória do resolvedor foi identificado um alto consumo de memória, assim sendo necessário realizar uma otimização no uso de memória ou verificar se está ocorrendo vazamento de memória pelo resolvedor.

Com a implementação da funcionalidade de propagação, uma melhoria para esta funcionalidade é aceitar uma lista com a ordem de valoração e o seu valor de verdade de modo a visualizar a propagação destas valorações pela fórmula.

Referências

ANDREI, S. et al. Automatic debugging of real-time systems based on incremental satisfiability counting. *IEEE Transactions on Computers*, v. 55, n. 7, p. 830–842, 2006. Citado na página 12.

BACCHUS, F.; WALSH, T. A non-CNF DIMACS style. 2005. Citado na página 17.

BIERE, A. The AIGER And-Inverter Graph (AIG) Format Version 20070427. Johannes Kepler University, Austria, 5 2007. Disponível em: <http://fmv.jku.at/papers/Biere-FMV-TR-07-1.pdf>. Citado 7 vezes nas páginas 25, 26, 27, 28, 29, 41 e 56.

BIERE, A.; HELJANKO, K. AIGER 1.9 And Beyond. Johannes Kepler University, Austria, 2011. Disponível em: <http://fmv.jku.at/papers/BiereHeljankoWieringa-FMV-TR-11-2.pdf>. Citado na página 26.

BITTENCOURT, M. C. de. *Comparing Different And-Inverter Graph Data Structures*. 20-30 p. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul, 2018. Citado 2 vezes nas páginas 26 e 29.

BRUMMAYER, R.; BIERE, A. Local Two-Level And-Inverter Graph Minimization without Blowup. 2006. Citado na página 18.

BRYAN, D. The ISCAS'85 benchmark circuits and netlist format. 1985. Citado na página 17.

BÜNING, H. et al. *Propositional Logic: Deduction and Algorithms*. [S.l.]: Cambridge University Press, 1999. 2 p. (Cambridge Tracts in Theoretical Computer Science). ISBN 9780521630177. Citado 2 vezes nas páginas 14 e 15.

CNF2AIG. 2008. Disponível em: <http://fmv.jku.at/cnf2aig/index.html>. Citado na página 39.

COOK, S. A. The complexity of theorem-proving procedures. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. New York, NY, USA: Association for Computing Machinery, 1971. (STOC '71), p. 151–158. ISBN 9781450374644. Disponível em: <https://doi.org/10.1145/800157.805047>. Citado na página 12.

DARWICHE, A.; PIPATSRISAWAT, K. Complete Algorithms. In: BIERE, A. et al. (Ed.). *Handbook of satisfiability*. [S.l.]: IOS press., 2009. p. 3–74. Citado 6 vezes nas páginas 13, 20, 21, 22, 24 e 25.

DAVIS, M.; LOGEMANN, G.; LOVELAND, D. A machine program for theorem-proving. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 5, n. 7, p. 394–397, jul 1962. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/368273.368557>. Citado na página 21.

DAVIS, M.; PUTNAM, H. A computing procedure for quantification theory. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 7, n. 3, p. 201–215, jul 1960. ISSN 0004-5411. Disponível em: <https://doi.org/10.1145/321033.321034>. Citado na página 20.

DRECHSLER, R.; JUNTTILA, T.; NIEMELA, I. Non-clausal SAT and ATPG. In: BIERE, A. et al. (Ed.). *Handbook of satisfiability*. [S.l.]: IOS press., 2009. p. 655–694. Citado 2 vezes nas páginas 13 e 16.

FRANCO, J.; MARTIN, J. A History of Satisfiability. In: BIERE, A. et al. (Ed.). *Handbook of satisfiability*. [S.l.]: IOS press., 2009. p. 3–74. Citado 3 vezes nas páginas 12, 19 e 21.

GOMES, C. P. et al. Satisfiability Solvers. In: HARMELEN, F. van; LIFSCHITZ, V.; PORTER, B. (Ed.). *Handbook of Knowledge Representation*. [S.l.]: Elsevier Science, 2007. p. 89–134. Citado 3 vezes nas páginas 13, 19 e 20.

JAIN, H.; CLARKE, E. M. Efficient SAT Solving for Non-Clausal Formulas Using DPLL, Graphs, and Watched Cuts. 2009. Citado na página 16.

KHURSHID, S.; MARINOV, D. Testera: Specification-based testing of java programs using sat. *Autom. Softw. Eng.*, v. 11, p. 403–434, 10 2004. Citado na página 13.

KLEMENT, K. C. Propositional Logic. In: FIESER, J.; DOWNDEN, B.; BéZIAU, J.-Y. (Ed.). *Internet Encyclopedia of Philosophy*. [s.n.], 2004. Disponível em: <https://iep.utm.edu/prop-log>. Citado na página 14.

LAPLANTE, P. A.; DEFRANCO, J. F. Software Engineering of Safety-Critical Systems: Themes From Practitioners. *IEEE Transactions on Reliability*, v. 66, n. 3, p. 825–836, 2017. Citado na página 12.

LINGAPPAN, L.; JHA, N. K. Satisfiability-based automatic test program generation and design for testability for microprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 15, n. 5, p. 518–530, 2007. Citado na página 12.

LYNCE, I.; MARQUES-SILVA, J. Efficient haplotype inference with boolean satisfiability. *Proceedings of the National Conference on Artificial Intelligence*, v. 1, 01 2006. Citado na página 12.

MARQUES-SILVA, J.; LYNCE, I.; MALIK, S. CDCL Solvers. In: BIERE, A. et al. (Ed.). *Handbook of satisfiability*. [S.l.]: IOS press., 2009. p. 131–154. Citado 2 vezes nas páginas 21 e 35.

MCCLARY, D. C. *Olympic Pipe Line accident in Bellingham kills three youths on June 10, 1999*. 2003. Disponível em: <https://historylink.org/File/5468>. Citado na página 12.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis. Universidade da Califórnia, Berkeley - CA, 2006. Disponível em: <https://people.eecs.berkeley.edu/~brayton/publications/2006/dac06_rwr.pdf>. Citado 2 vezes nas páginas 18 e 19.

MOSKEWICZ, M. et al. Chaff: engineering an efficient sat solver. In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. [S.l.: s.n.], 2001. p. 530–535. Citado na página 22.

MOUHOUB, M.; SADAOUI, S. Solving incremental satisfiability. *International Journal on Artificial Intelligence Tools*, v. 16, p. 139–147, 02 2007. Citado na página 20.

NETO, W. L. *Exact Multi-Level Benchmark Circuit Generation for Logic Synthesis Evaluation*. 24–25 p. Dissertação (Mestrado em Microeletrônica) — Universidade Federal do Rio Grande do Sul, 2018. Citado na página 18.

RIBAS, B. C. *Satisfatibilidade não-clausal restrita às variáveis de entrada*. Dissertação (Mestrado em Informática) — Universidade Federal do Paraná, 2011. Citado 9 vezes nas páginas 12, 17, 22, 24, 32, 34, 35, 36 e 37.

ROTH, C. H.; KINNEY, L. L. *Fundamentals of Logic Design*. 7. ed. [S.l.]: Cengage Learning, 2014. 336-369 p. Citado 2 vezes nas páginas 27 e 40.

RUSSELL, S. J.; NORVIG, P.; CHANG, M.-W. Artificial intelligence: a modern approach. In: _____. 3. ed. Upper Saddle River, New Jersey: Pearson, 2010. cap. 11, p. 401–436. Citado na página 12.

SATCompetitions. In: The International SAT Competition . [s.n.], 2005. Disponível em: <http://www.satcompetition.org/>. Citado na página 20.

SAT-Race 2010. In: SINZ, C. (Ed.). *13th International Conference on Theory and Applications of Satisfiability Testing*. Edimburgo, Escócia: [s.n.], 2010. Disponível em: <https://baldur.iti.kit.edu/sat-race-2010/index.html>. Citado 2 vezes nas páginas 39 e 46.

SMITH, A. et al. Fault diagnosis and logic debugging using boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 24, n. 10, p. 1606–1621, 2005. Citado na página 12.

THIFFAULT, C.; BACCHUS, F.; WALSH, T. Solving Non-clausal Formulas with DPLL search. 2004. Citado 2 vezes nas páginas 16 e 19.

WANG, Y.; DING, H. Construction of the graphical propositional logic. *TELKOMNIKA Indonesian Journal of Electrical Engineering*, v. 11, 08 2013. Citado na página 15.

WARNERS, J. P. A Linear-Time Transformation of Linear Inequalities into Conjunctive Normal Form. Centrum Wiskunde & Informatica, Universidade de Amsterdã - Países Baixos, p. 2, 1996. Citado na página 16.

YOUNESS, H. et al. An effective sat solver utilizing aco based on heterogenous systems. *IEEE Access*, v. 8, p. 102920–102934, 2020. Citado na página 47.

Apêndices

APÊNDICE A – LIAMFSAT

A.1 Como compilar e executar o resolvedor

A.1.1 Compilação

De modo a conseguir utilizar o resolvedor LIAMFSAT, é recomendável a utilização de alguma distribuição Linux com os seguintes pacotes instalados:

- Make;
- GCC.

Para compilar o resolvedor é necessário estar no seu diretório raiz através do terminal e inserir o seguinte comando:

```
make
```

Existem dois comandos utilizados para remover do diretório os arquivos criados no processo de compilação.

1. O primeiro comando realiza uma remoção superficial, removendo os arquivos de compilação e executáveis do resolvedor.
2. O segundo comando realiza uma remoção profunda, removendo os arquivos de compilação, executáveis, bibliotecas e ferramentas.

   Os comandos são:

   ```
   make clean
   ```

   e

   ```
   make dist-clean
   ```

A.1.2 Uso

Após o processo de compilação, são gerados os executáveis para utilizar as diversas funcionalidades e ferramentas do LIAMFSAT. Para executar o LIAMFSAT é necessário estar no diretório raiz do resolvedor através do terminal e utilizar o seguinte comando:

```
./liamfsat <funcionalidade> <diretório do arquivo de
    entrada> [diretório de saída]
```

ou

```
./liamfsat <ferramenta> <caminho até o arquivo de entrada>
    [diretório de saída]
```

Nos próximos tópicos são descrito as funcionalidades e ferramentas disponíveis para utilização.

A.1.3 Funcionalidades

- **solver**: resolve o Problema de Satisfatibilidade aplicado a uma fórmula na representação ISCAS ou AIG. Utiliza apenas o parâmetro que indica o arquivo de entrada;
- **graphviz**: gera um arquivo .dot com o grafo da fórmula passada no arquivo de entrada;
- **reinstanciate**: mostra informações sobre a fórmula, como quantidade de entradas, cláusulas e tempo de travessia. Utiliza apenas o parâmetro que indica o arquivo de entrada;
- **propagation**: gera diversos arquivos .dot com o processo de propagação na fórmula em cada processo de valoração parcial ou retrocesso. A fórmula é passada no arquivo de entrada no segundo parâmetro e o diretório é passado no terceiro parâmetro.

A.1.4 Ferramentas

Todos os comandos utilizam tanto o parâmetro de entrada quanto o parâmetro de saída.

- **aigtoaig**: esta ferramenta funciona apenas para a representação AIG. Caso o arquivo esteja no formato binário, o arquivo convertido estará no formato ASCII. Caso o arquivo esteja no formato ASCII, o arquivo convertido estará no formato binário;
- **aigtoiscas**: converte uma fórmula na representação AIG para a representação ISCAS;
- **iscastoaig**: converte uma fórmula na representação ISCAS para a representação AIG.

Anexos

ANEXO A – AIGER

A.1 Funções em C para codificar para binário e decodificar para ASCII os deltas de operadores ∧ em arquivos AIGER

```
1  unsigned char getnoneofch(FILE *file) {
2      int ch = getc(file);
3
4      if(ch != EOF)
5          return ch
6
7      fprintf(stderr, ``*** decode: unexpected EOF\n'');
8      exit(1);
9  }
10
11 unsigned decode(FILE *file) {
12     unsigned x = 0, i = 0;
13     unsigned char ch;
14
15     while ((ch = getnoneofch(file)) & 0x80)
16         x |= (ch & 0x7f) << (7 * i++);
17
18     return x | (ch << (7 * i));
19 }
20
21 void encode(FILE *file, unsigned x) {
22     unsigned char ch;
23
24     while(x & ~0x7f) {
25         ch = (x & 0x7f) | 0x80;
26         putc(ch, file);
27         x >>= 7;
28     }
29
30     ch = x;
31     putc(ch, file);
32 }
```

Código A.1 – Código parte da biblioteca AIGER. Autor: (BIERE, 2007)

[1] Licença do código fonte da biblioteca AIGER está disponível em: https://github.com/arminbiere/aiger/blob/master/LICENSE.

[2] Código fonte completo da biblioteca AIGER está disponível em: https://github.com/arminbiere/aiger.