Trabalho Final: Algoritmos para o problema de Satisfatibilidade.

00341816 - Francisco Pegoraro Etcheverria
00342346 - Guilherme d'Ávila Pinheiro
00341853 - Henrique Lorentz Trein
00337514 - Luiz Fernando Pascoal de Oliveira

1. INTRODUÇÃO

O problema de satisfatibilidade booleana questiona se há pelo menos uma combinação de valoração das variáveis para a qual a fórmula retorne verdadeiro. Se este é o caso, diz-se que a fórmula é satisfazível. Seguindo a convenção que faz referência a esse problema, chamaremos-o de SAT (abreviação de "satisfiability" da língua inglesa). Dessa forma, "SAT solvers" são algoritmos que estabelecem satisfatibilidade, os quais normalmente recebem uma fórmula como entrada e retornam "SAT" se foi encontrada uma combinação de valores que satisfaz a fórmula ou "UNSAT" caso contrário. Além dessas opções, às vezes, podem não conseguir determinar uma resposta. SAT é um problema central na Ciência da Computação dito como o primeiro problema NP-Completo (da língua inglesa "nondeterministic polynomial-time complete") e sua prova é sustentada pelo teorema de Cook-Levin. Problemas dessa classe são definidos como intratáveis, pois não conseguem ser solucionados a partir de algoritmos polinomiais, mas podem ter seus resultados verificados de maneira polinomial. O problema de SAT possui diversas implementações eficientes e algumas das mais reconhecidas são: DPLL, CDCL, WalkSAT, GRASP, Glucose e Z3 (criado pela Microsoft Research).

2. DESENVOLVIMENTO

Os "SAT solvers" a serem expostos no trabalho são variantes apresentadas pelo livro "Logic in Computer Science: Modelling and reasoning about systems" de M. Huth e M. Ryan, do método de Stalmarck, um solucionador de satisfatibilidade patenteado na Suécia e nos Estados Unidos. Ambos os solucionadores prezam pela eficiência, mas como consequência, acabam por não serem completos.

2.1 Solucionador Linear

Procedimento baseado na propagação de marcas entre as subfórmulas:

Passo 1: Gerar uma fórmula semanticamente equivalente e com os mesmos átomos proposicionais T(φ) a partir de φ, com apenas negação e conjunção de literais i.e. T(φ) ::= p | (~φ) | (φ ∧ φ). Isso garante que para todas as valorações em que φ é verdadeira, T(φ) também é. Traduz-se da seguinte maneira:

$T(p) = p$
$T(\varphi^1 \wedge \varphi^2) = T(\varphi^1) \wedge T(\varphi^2)$
$T(\varphi^1 \rightarrow \varphi^2) = \sim(T(\varphi^1) \wedge \sim T(\varphi^2))$
$T(\sim\varphi) = \sim T(\varphi)$
$T(\varphi^1 \vee \varphi^2) = \sim(\sim T(\varphi^1) \wedge \sim T(\varphi^2))$

Passo 2: Construir a árvore sintática da fórmula.

Passo 3: Construir um DAG (acrônimo da língua inglesa para Directed Acyclic Graph, ou seja, Grafo Acíclico Direcionado) a partir da árvore sintática e seus nós compartilhados.

Passo 4: Identificação do DAG no formato "X:Y", com X indicando a ordem do raciocínio (geralmente, não é única) e Y para a valoração do nó. O nó superior do grafo deve receber verdadeiro, uma vez que representa o valor de verdade da fórmula. Segue-se a lógica do topo para a base, marcando os nós com seus respectivos valores de verdade usando as leis de forçamento: 1) **Negação. 1.1** '¬t' para ¬ verdadeira e seu nodo

imediatamente abaixo falso. **1.2** '¬f' para ¬ falsa e seu nodo imediatamente abaixo verdadeiro. 2) **Conjunção: 2.1** '∧te' para conjunção verdadeira força termos da conjunção verdadeiros. **2.2** '∧ti' para termos da conjunção verdadeiros forçam conjunção verdadeira. **2.3** '∧fl' para termo da conjunção esquerdo falso força conjunção falsa. **2.4** '∧fr' para termo da conjunção direito falso força conjunção falsa. **2.5** '∧fll' para conjunção falsa e termo da conjunção esquerdo verdadeiro força termo da conjunção direito falso **2.6** '∧frr' para conjunção falsa e termo da conjunção direito verdadeiro força termo da conjunção esquerdo falso.

Passo 5: Uma segunda fase do procedimento recalcula as marcas de todos os nós de baixo para cima, se e somente se as marcas resultantes coincidirem com as que calculamos antes, teremos encontrado uma prova.

O solucionador SAT também pode ser utilizado para checar a validade de sequentes, transformando-os em uma implicação negada e gerando a fórmula semanticamente equivalente válida para o SAT. A fórmula é negada pois o algoritmo apenas obtém uma valoração para qual a fórmula é satisfeita, portanto é prático provar o sequente demonstrando que não é possível obter um contra exemplo, ou seja, quando a negação da fórmula resulta em uma contradição e portanto é insatisfazível.

Dado um sequente como:

$(\neg\varphi^1 \rightarrow \neg\varphi^2) \wedge \varphi^2 \vdash \varphi^1$

Utiliza-se uma implicação negada:

$\neg((\neg\varphi^1 \rightarrow \neg\varphi^2) \wedge \varphi^2 \rightarrow \varphi^1)$

Então os passos definidos anteriormente são utilizados normalmente.

Conclusão: Se em um mesmo nodo tivermos o forçamento de T e F (verdadeiro e falso, respectivamente), temos um problema. Tais contradições implicam que todas as fórmulas T(φ) cujo DAG é igual a esse em questão, não são satisfatíveis. Este solucionador SAT tem um tempo de execução linear no tamanho do DAG para T(φ), ou seja, nossa tradução prévia de φ causa apenas um aumento linear. Essa linearidade vem com um custo alto: nosso solucionador linear falha para todas as fórmulas na forma ¬(φ1 ∧ φ2).

2.2 Solucionador Cúbico

Como visto anteriormente, o solucionador linear não é suficiente para determinar se certas fórmulas são satisfazíveis. Ao aplicarmos o solucionador linear, podemos encontrar marcas contraditórias, e concluir que as fórmulas representadas pelo DAG são insatisfazíveis, ou encontramos marcas consistentes, e a fórmula é satisfazível. Porém, há uma terceira possibilidade: todas marcas são consistentes, mas nem todos os nodos foram marcados. O solucionador cúbico busca melhorar essa análise:

Passo 1: Após ser aplicado o solucionador linear, se ainda existirem nodos não marcados, escolha um nodo não marcado qualquer *n,* e faça dois testes:

1.1: Determine que marcas *temporárias* são forçadas quando *n* for marcado como verdadeiro.

1.2: Determine que marcas temporárias são forçadas quando *n* for marcado como falso.

Passo 2: Se em ambos os testes houverem contradições, o algoritmo para, e determinamos que T(φ) é insatisfazível. Se não, todos os nodos que receberam a mesma valoração *temporária* em ambos testes passam a ter aquela valoração *permanente*, e testamos os nodos ainda não marcados novamente. O processo se repete até que:

2.1: Existem marcas *permanentes* contraditórias. Concluímos que T(φ) é insatisfazível.

2.2: Todos os nodos foram marcados e não existem contradições. Concluímos que T(φ) é satisfazível.

2.3: Todos os nodos não marcados foram testados, mas não existe valoração consistente entre ambas valorações para o nodo. Neste caso, terminamos a análise sem determinar se T(φ) é satisfazível.

Este solucionador tem tempo de execução cúbico no tamanho do DAG para T(φ). Um fator é devido ao uso do solucionador linear para cada teste. Outro fator é a necessidade de testar todo nodo não marcado. Outro é o fato de que a adição de uma marca permanente faz com que tenhamos de testar todos nodos não marcados novamente.

2.3 Otimização do Solucionador Cúbico

Considere o estado de um DAG logo após explorarmos as consequências de uma marca temporária em um nó de teste.

1. Se esse estado - marcas permanentes mais temporárias - contiver restrições contraditórias, podemos apagar todas as marcas temporárias e marcar permanentemente o nó de teste com a marca dual do seu teste. Ou seja, se marcar o nó n com v resultou em uma contradição, ele receberá uma marca permanente $\bar{v}$, onde $\bar{T} = F$ e $\bar{V} = T$; caso contrário,
2. Se esse estado conseguir marcar todos os nós com restrições consistentes, relatamos essas marcações como uma prova de satisfatibilidade e encerramos o algoritmo.

3. EXEMPLOS

3.1 Solucionador linear:

1. φ = p ∧ ¬(q ∨ ¬p), transforma-se para T(φ) desta maneira:

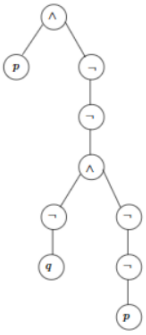
(q ∨ ¬p) = ¬(¬(q) ∧ ¬(¬p))

p = p

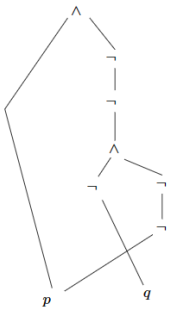
p∧
¬(¬(¬(q)∧¬(¬p)))=(p)∧(¬(¬(¬(q)∧¬(¬p))))

simplificando, p ∧ ¬¬(¬q ∧ ¬¬p).

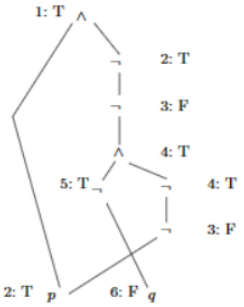
2. Árvore sintática:



3. DAG:



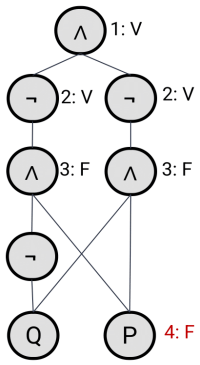
4. Marcações:



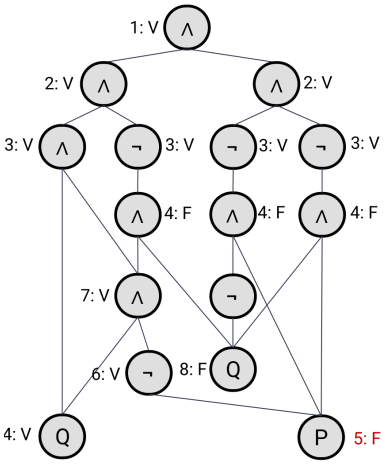
5. Conclusão:

Como não existem nodos que não foram marcados e nem contradições, tal fórmula é **satisfazível.**

3.2 Solucionador cúbico:



No exemplo acima, Q é marcado temporariamente com V e F e em ambos os casos P é falso, portanto P é marcado permanentemente. No entanto, o DAG continua indeterminado.



Nesse exemplo o método do solucionador cúbico é suficiente para verificar que o DAG é satisfazível.

4. APLICAÇÕES PRÁTICAS E TEÓRICAS

O problema de satisfatibilidade está associado a diversas aplicações computacionais fundamentais, assim existem diversos solucionadores desse tipo de problema. “SAT solvers” têm aplicações em: inteligência artificial, modelando redes neurais binárias, modelagem de árvores de decisão binárias, pode ser adaptado com variáveis booleanas que representam cores e restrições para a repetição dessas cores entre “vizinhos” em problemas de coloração de grafos e é um meio de estudo útil para aplicações de agendamento, evitando conflitos de horários, por exemplo. É prudente mencionar que os solucionadores SAT estão próximos de outros tipos de problemas importantes além da satisfabilidade, como a contagem dos literais distintos que satisfazem uma fórmula (Contagem de Modelos) ou a valoração para minimizar cláusulas inválidas (Max-SAT). Além de tudo isso, possuem aplicações na otimização no design de circuitos e chips, sistemas de segurança e otimização de compiladores.

5. BIBLIOGRAFIA

M. Huth and M. Ryan. Logic in Computer Science: Modelling and reasoning about systems. Cambridge University Press, 2004.
S. Prince. SAT Solvers I: Introduction and applications. Borealis AI, 2020.
M. Finger. SAT Solvers, A Brief Introduction. Instituto de Matemática e Estatística - USP