PROBLEMA DE SATISFAÇÃO BOOLEANA (SAT)

PROJETO E ANÁLISE DE ALGORITMOS

Discente: Lauryane Santos Siqueira
Docente: Leonardo Nogueira Matos

PROCC
Programa de Pós-Graduação em Ciência da Computação/UFS

UNIVERSIDADE FEDERAL DE SERGIPE

POR QUE ESTUDAR SAT?

- Um dos problemas mais fundamentais da Computação Teórica
- Primeiro problema provado NP-Completo (Cook–Levin)
- Base para áreas como: verificação formal, IA, compiladores, criptografia
- Muitos problemas reais podem ser reduzidos a SAT

Stephen Cook
Fonte: Alchetron. Disponível em: https://alchetron.com/Stephen-Cook

O QUE É O PROBLEMA SAT?

- "Dada uma fórmula booleana, existe uma atribuição de valores (TRUE/FALSE) às variáveis que torna toda a fórmula verdadeira?"
- **Exemplo simples**:
  - $(A \vee B) \wedge (\neg A \vee C)$
  - Pergunta: existe uma atribuição que satisfaça?

ELEMENTOS PRINCIPAIS

- Variáveis: x_1, x_2, ..., x_n
- Literais: variável ou negação (x_i ou ¬x_i) ; “verdadeiro” (1) ou “falso” (0)
- Cláusulas ou Operadores de literais:

~ negação (not, inversão, -). Inverte o valor da variável.
tabela verdade:

x	~x
0	1
1	0

^ conjunação (and, e, etc). Tem resultado verdadeiro apenas quando ambos operandos são verdadeiros.
tabela:

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

v disjunção (or, ou, etc). Tem resultado falso apenas quando ambos operandos são falsos.

x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1

FÓRMULA

- As fórmulas booleanas combinam variáveis e operadores. Também é possível utilizar parênteses para determinar a ordem de algumas operações.
- Exemplo:

$$F(x_0 , x_1 , x_2) = (\sim x_0 \ \wedge x_1) \vee (x_0 \ \wedge x_2) \vee (x_1 \wedge x_2)$$

Usando a notação “.” para conjunção e “+” para disjunção:

$$F(x_0 , x_1 , x_2) = (\sim x_0 \ . x_1) + (x_0 \ . x_2) + (x_1 . x_2)$$

O PROBLEMA DO SAT ESTÁ EM DETERMINAR SE UMA FÓRMULA BOOLEANA QUALQUER POSSUI UM CONJUNTO DE VALORES VERDADE PARA SUAS VARIÁVEIS QUE A TORNAM VERDADEIRA.

AVALIAÇÃO DO EXEMPLO

- Para a fórmula apresentada, uma atribuição possível é:

$$x_0 = 1, x_1 = 1, x_2 = 1 \rightarrow F = 1$$

- Logo, a instância do SAT é satisfatível (existe solução).
- Para N variáveis existem 2^n combinações possíveis.
- Ex.: fórmula com 3 variáveis → 8 linhas na tabela verdade.

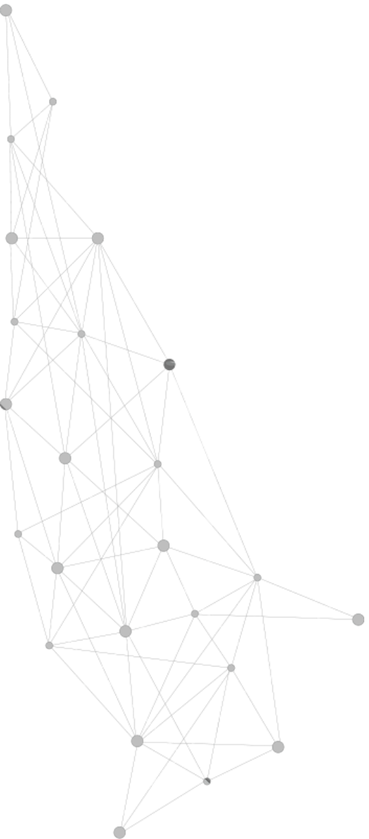
TABELA VERDADE DO EXEMPLO

- A fórmula só é verdadeira em algumas combinações.
- Ideia central: SAT busca “uma linha verdadeira” da tabela, mas sem precisar gerar a tabela inteira.

x_0	x_1	x_2	F	
0	0	0	0	
0	0	1	0	
0	1	0	1	
0	1	1	1	
1	0	0	0	
1	0	1	1	
1	1	0	0	
1	1	1	1	

CNF (FORMA NORMAL CONJUNTIVA)

- CNF = conjunção ($\wedge$) de cláusulas.
- Cada cláusula = disjunção ($\vee$) de literais.
- Literais = variáveis ou suas negações.
- Qualquer expressão booleana pode ser convertida para CNF.
- SAT geralmente é definido sobre fórmulas em CNF.



EXEMPLOS DE ENTRADAS DO PROBLEMA

- $Y = BC + \bar{A}\bar{B}\bar{C} + B\bar{C}$
- $F = (A+\bar{B}).(A + C)$
- $Z = (A + \sim A).(A.B + A.B.(\sim C))$
- $W = (A.\bar{A})(B + C)$

Saída do SAT:
“Existe alguma atribuição que satisfaz a fórmula?”
“Sim” ou “Não”, opcionalmente com a atribuição encontrada.

EXEMPLOS CONVERTIDOS PARA CNF

- $Y = (A + B + C) . (A + B + \bar{C}) . (\bar{A} + B + C) . (\bar{A} + B + \bar{C}) . (\bar{A} + \bar{B} + C)$
- $F = (A + B + C) . (A + B + \bar{C}) . (A + \bar{B} + C) . (A + \bar{B} + \bar{C})$
- $Z = (A + B + C) . (A + B + \bar{C}) . (A + \bar{B} + C) . (A + \bar{B} + \bar{C}) . (\bar{A} + B + C) . (\bar{A} + B + \bar{C})$

(Observação: Cook trabalhou explicitamente com CNF-SAT, agora o caso clássico.

EXEMPLOS DE RESULTADO (SATISFATÍVEL OU NÃO)

- Y: tem solução. Ex.: B = 1, C = 1, A = # (# representa que A pode ser tanto 0 ou 1).
- F: tem solução. Ex.: A = 1, B = 0, C = 1.
- Z: tem solução. Ex.: A = 1, B = 1, C = 0.
- W: não tem solução, pois $(A.\bar{A})=0$.

CLASSIFICAÇÃO DO PROBLEMA

(1/2) – Por que SAT está em NP

- Para estar em NP: precisamos verificar uma solução em tempo polinomial.
- Dada uma atribuição, basta avaliar a fórmula uma vez → tempo polinomial.
- Portanto: SAT ∈ NP.

(2/2) – NP-Difícil e NP-Completo

- **NP-Difícil:**

Todo problema de NP pode ser reduzido a SAT em tempo polinomial.
A ideia é que um algoritmo NP pode ser representado como uma sequência de escolhas e estas podem ser codificadas como variáveis booleanas.

- **NP-Completo:**

SAT é NP + NP-Difícil.

Curiosidade: Primeiro problema provado NP-Completo (Cook, Levin, anos 70).

ALGORITMO PARA SAT (BACKTRACKING) - PSEUDOCODIGO

backtrack(ϕ):

```
if φ = Ø: return True

if ε ∈ φ: return False

let x = pick_variable(φ)

return backtrack(φ|x) OR backtrack(φ|¬x)
```

- Estratégia: tentamos atribuições recursivamente.
- Poda ocorre quando alguma cláusula fica vazia (não pode ser satisfeita).

COMPLEXIDADE DO ALGORITMO

- No pior caso:

$$T(n) = 2T(n-1) \rightarrow O(2^n)$$

- Melhorias práticas:
  - heurísticas de seleção de variáveis;
  - poda;
  - aprendizado de cláusulas (no estilo DPLL/CDCL).
- Apesar das heurísticas, o pior caso continua exponencial (problema NP-Completo)

INSTÂNCIA DO PROBLEMA

Exemplos

```
1    Y:A,B,C;A,B,-C;-A,B,C;-A,B,-C;-A,-B,C
2    F:A,B,C;A,B,-C;A,-B,C;A,-B,-C
3    Z:A,B,C;A,B,-C;A,-B,C;A,-B,-C;-A,B,C;-A,B,-C
```

IMPLEMENTAÇÃO EM PYTHON

```python
# Funcao principal
def backtrack(phi, assignment=None):
    if assignment is None:
        assignment = {}

    # Caso 1: fórmula vazia → solução encontrada
    if phi == []:
        return assignment

    # Caso 2: cláusula vazia → contradição
    if any(clause == [] for clause in phi):
        return None

    # variável escolhida
    x = pick_variable(phi)

    # Tenta x = True
    new_assign = assignment.copy()
    new_assign[x] = True
    result = backtrack(reduce_formula(phi, x, True), new_assign)
    if result is not None:
        return result

    # Tenta x = False
    new_assign = assignment.copy()
    new_assign[x] = False
    return backtrack(reduce_formula(phi, x, False), new_assign)

```

Função Principal

IMPLEMENTAÇÃO EM PYTHON

```python
# Funcao auxiliar para escolher a proxima variavel
def pick_variable(phi):
    for clause in phi:
        for lit in clause:
            return lit.lstrip('-')  # remove o '-' se existir


# funcao auxiliar: reduzir a formula a partir da atribuicao da variavel atual
def reduce_formula(phi, var, value):
    new_formula = []

    for clause in phi:
        new_clause = []
        clause_satisfied = False

        for lit in clause:
            if lit == var:  # literal positivo
                if value is True:
                    clause_satisfied = True
                    break
                else:
                    continue  # literal falso → ignora
            elif lit == f"-{var}":  # literal negativo
                if value is False:
                    clause_satisfied = True
                    break
                else:
                    continue
            else:
                new_clause.append(lit)

        # se a cláusula foi satisfeita, simplesmente a removemos
        if clause_satisfied:
            continue

        new_formula.append(new_clause)

    return new_formula
```

Redução da Fórmula

IMPLEMENTAÇÃO EM PYTHON

```python
# Resolução das instâncias apresentadas
# o formato de cada problema é -> nome_da_formula:disjunção1;disjunção2;...
# cada disjunção é no formato: Literal1,Literal2, ...
# cada literal é uma variável ou sua negação: A,-B,C ...
# Não é obrigatório usar A, B, C, etc para variáveis. (Ex: x,y,z,etc)

# Ler o arquivos de exemplos:

with open('exemplos.txt', 'r+') as exemplos:
    for line in exemplos:
        nome_formula, disjuncoes = line.split('\n')[0].split(':')
        disjuncoes = [y.split(",") for y in disjuncoes.split(";")]

        aux = [f"({' + '.join(x)})" for x in disjuncoes]
        print(f"{nome_formula} = {' . '.join(aux)}")

        solucao = backtrack(disjuncoes)

        if solucao is None:
            print("Insatisfatível")
        else:
            # imprime a solução formatada
            for var, val in solucao.items():
                print(f"{var} = {val}")
        print()
```

Função Aplicada aos Exemplos

SOLUÇÃO

```
Y = (A + B + C) . (A + B + -C) . (-A + B + C) . (-A + B + -C) . (-A + -B + C)
A = True
B = True
C = True

F = (A + B + C) . (A + B + -C) . (A + -B + C) . (A + -B + -C)
A = True

Z = (A + B + C) . (A + B + -C) . (A + -B + C) . (A + -B + -C) . (-A + B + C) . (-A + B + -C)
A = True
B = True


Process finished with exit code 0
```

REFERÊNCIAS

- Levitin, cap. 11.3
- CIS 189 – Lecture 3: Algorithms for SAT
- Handbook of Satisfiability (Biere, 2009)
- Eén & Sörensson (2004) – MiniSAT