UFS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Atividade:

Problema da Partição de Conjuntos (Set Partition Problem)

Professor:

Dr. Leonardo Nogueira Matos

Equipe

Francisco Farias Gomes

Silas Lopes Santos Silva Amancio do Vale

Luciano Torres Marques

Ramon Adller de Santana

Definição do Problema

Definição do Problema

O Problema da Partição de Conjuntos é um problema clássico de decisão que busca determinar se um conjunto finito de números inteiros positivos pode ser dividido em dois subconjuntos disjuntos cuja soma dos elementos seja exatamente igual.

Definição do Problema

Exemplo

P: Ordenar uma lista usando o algoritmo **Quicksort**, com complexidade $O\,(n \log n)$.

NP: Verificar se um ciclo em um grafo é hamiltoniano, ou seja, se passa uma única vez por todos os vértices.

Definição do Problema

Definição

Dado um conjunto $S = \{a_1, a_2, ..., a_n\}$ de inteiros positivos, determine se existe uma partição em dois subconjuntos disjuntos S_1 e S_2 tais que:

Definição do Problema

Definição

Dado um conjunto **S** = {a_1, a_2, ..., a_n} de inteiros positivos, determine se existe uma partição em dois subconjuntos disjuntos **S_1** e **S_2** tais que:

$$\sum S_1 = \sum S_2$$

Definição do Problema

Definição

Ou seja: a soma dos elementos de $\mathbf{S_1}$ é igual à soma dos de $\mathbf{S_2}$.

Exemplo:

$$S = \{1, 2, 3, 6\} \longrightarrow \quad S_1 = \{1, 2, 3\},\ S_2 = \{ 6 \} \longrightarrow \text{soma} = 6$$

Definição do Problema

Definição

A classe **NP** é a classe de problemas de decisão que podem ser resolvidos por algoritmos polinomiais não-determinísticos. Essa classe de problemas é chamada polinomial não-determinístico.

Definição do Problema

Definição

Ou seja, em termos de verificação de tempo polinomial, Existe um verificador determinístico que, dado:

Definição do Problema

Definição

Ou seja, em termos de verificação de tempo polinomial, Existe um verificador determinístico que, dado:

- Uma instância **I** do problema
- Um certificado **c** (de tamanho polinomial em |**I**|),

decide em tempo polinomial se **c** é uma solução válida para **I**.

Como provar que é NP-completo?

Como provar que é NP-completo

Para provar que o problema da Partição (**Partition**) é NP-completo, é necessário demonstrar dois passos fundamentais:

Como provar que é NP-completo

1. Partition $\in$ NP, ou seja, uma solução candidata pode ser verificada em tempo polinomial.

   certificado: um subconjunto S_1;

   verificação: somar S_1 e $S \setminus S_1$ em $O(n)$.

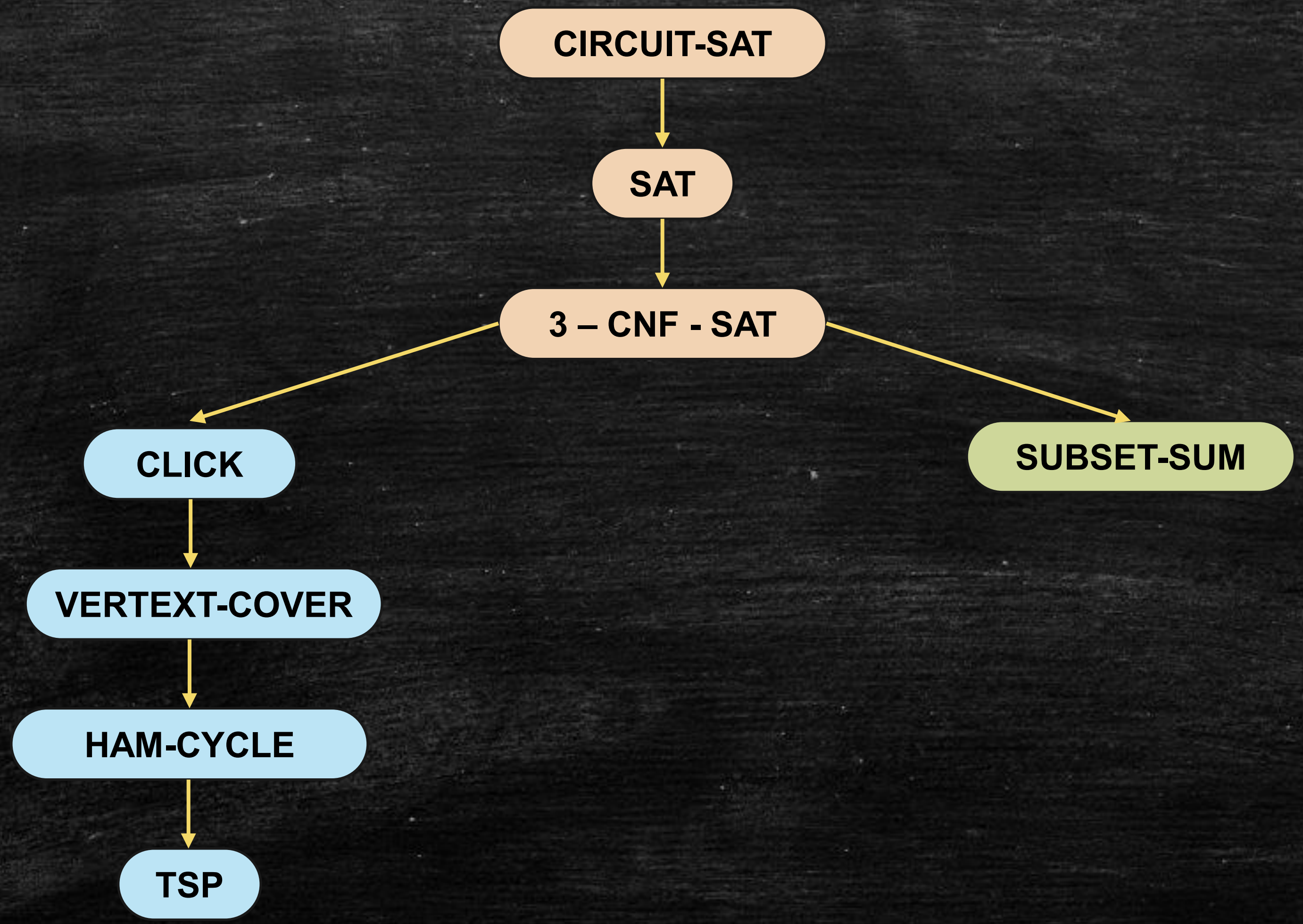
Como provar que é NP-completo

2. Partition é NP-hard, provado por redução polinomial de um problema já conhecido como NP-completo — no caso, SUBSET-SUM. A redução transforma uma instância de SUBSET-SUM $\langle T, K \rangle$ em uma instância de PARTITION ao construir

$$S = T \cup \{ \sigma - 2K \}$$

Como provar que é NP-completo

Estrutura das provas de NP-completude de CIRCUIT-SAT.



Importanica

Importanica

A NP-completude importa porque centenas de problemas reais — como roteamento de veículos, escalonamento de tarefas, alinhamento de sequências em bioinformática e quebra de criptografia — são NP-completos, e não possuímos algoritmos polinomiais para nenhum deles, nem provavelmente teremos, caso $P \neq NP$.

Importanica

As reduções polinomiais permitem provar a dureza de novos problemas ao mostrá-los equivalentes a outros já conhecidos, e reutilizar soluções aproximadas ou heurísticas entre eles; além disso, pequenas variações de problemas conhecidos geram novos NP-completos, reforçando que a teoria da NP-completude não é apenas acadêmica, mas um guia prático para lidar com a intratabilidade computacional em aplicações do mundo real.

Algoritmo de Partição por Backtracking

Algoritmo de Partição por Backtracking

Problema da Partição de Conjuntos Solução por Backtracking (Força Bruta Otimizada), Baseado nas teorias de Levitin e Cormen.

Algoritmo de Partição por Backtracking

```
def partition_backtracking(S):
    S = sorted(S, reverse=True)
    total = sum(S)

    if total % 2 != 0:
        return False, [], []

    target = total // 2

    def bt(pos, sum1, grupo1):
        if pos == len(S):
            return sum1 == target, grupo1[:]

        num = S[pos]
```

Algoritmo de Partição por Backtracking

```
if sum1 + num <= target:
    grupo1.append(num)
    encontrou, solucao = bt(pos + 1, sum1 + num, grupo1)
    if encontrou:
        return True, solucao
    grupo1.pop()

remaining = sum(S[pos+1:])
if sum1 + remaining >= target:
    encontrou, solucao = bt(pos + 1, sum1, grupo1)
    if encontrou:
        return True, solucao

return False, []
```

Algoritmo de Partição por Backtracking

```
    encontrou, parte1 = bt(0, 0, [])
    if not encontrou:
        return False, [], []

    parte2 = [x for x in S if x not in parte1]
    return True, sorted(parte1), sorted(parte2)

if __name__ == "__main__":
    testes = [
        [1, 5, 11, 5],
        [1, 2, 3, 6],
        [1, 2, 5],
        [3, 1, 1, 2, 2, 1],
        [4, 5, 6, 7, 8]
    ]
```

Algoritmo de Partição por Backtracking

```
for i, conjunto in enumerate(testes, 1):
    print(f"Teste {i}: {conjunto} → soma = 
{sum(conjunto)}")
    ok, A, B = partition_backtracking(conjunto)
    if ok:
        print(f"   Partição: {A} | {B}  →  {sum(A)} = 
{sum(B)}")
    else:
        print("   Impossível particionar")
    print()
```

Algoritmo de Partição por Programação Dinâmica

Algoritmo por Programação Dinâmica

Problema da Partição de Conjuntos Solução por Programação Dinâmica Baseado nas teorias de Levitin e Cormen

Algoritmo por Programação Dinâmica

```python
def partition_problem_dp(S):
    n = len(S)
    total_sum = sum(S)

    if total_sum % 2 != 0:
        return False, [], []


    target = total_sum // 2

    DP = [[False] * (target + 1) for _ in range(n + 1)]

    for i in range(n + 1):
        DP[i][0] = True
```

Algoritmo por Programação Dinâmica

```
for i in range(1, n + 1):
    for j in range(target + 1):

        DP[i][j] = DP[i-1][j]

        if j >= S[i-1]:

            DP[i][j] = DP[i][j] or DP[i-1][j - S[i-1]]


if not DP[n][target]:
    return False, [], []
```

Algoritmo por Programação Dinâmica

```
subset1 = []
subset2 = []
i, j = n, target

while i > 0:
    if DP[i-1][j]:
        subset2.append(S[i-1])
    else:
        subset1.append(S[i-1])
        j -= S[i-1]
    i -= 1

return True, subset1, subset2
```

Algoritmo por Programação Dinâmica

Podemos adotar uma solução mais eficiente para solucionar o Problema da Partição de Conjuntos Solução por Programação Dinâmica Baseado nas teorias de Levitin e Cormen

Algoritmo por Programação Dinâmica

```python
def partition_problem_dp_otimizado(S):
    total_sum = sum(S)
    if total_sum % 2 != 0:
        return False, [], []

    target = total_sum // 2
    DP = [False] * (target + 1)
    DP[0] = True

    for num in S:
        for j in range(target, num - 1, -1):
            if DP[j - num]:
                DP[j] = True

    return DP[target], [], []
```

Algoritmo por Programação Dinâmica

```
if __name__ == "__main__":
    print("="*70)
    print("PARTIÇÃO DE CONJUNTOS - PROGRAMAÇÃO DINÂMICA (Cormen p. 404)")
    print("="*70)


    exemplos = [
        [1, 5, 11, 5],
        [1, 2, 5],
        [3, 1, 1, 2, 2, 1],
        [2, 3, 7, 8, 10]
    ]
```

Algoritmo por Programação Dinâmica

```
for i, conjunto in enumerate(exemplos, 1):
        print(f"\nExemplo {i}: {conjunto} → Soma total = {sum(conjunto)}")
        existe, A, B = partition_problem_dp(conjunto)
        if existe:
            print(f"Partição encontrada!")
            print(f"   Subconjunto 1: {A} → soma = {sum(A)}")
            print(f"   Subconjunto 2: {B} → soma = {sum(B)}")
        else:
            print("Impossível particionar.")

        print(f"   [Otimizado] Existe partição? {'Sim' if partition_problem_dp_otimizado(conjunto)[0] else 'Não'}")
```

Atividade:

Problema da Partição de Conjuntos (Set Partition Problem)

UFS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO