Pontifícia Universidade Católica de Minas Gerais
Instituto de Ciências Exatas e Informática
Departamento de Engenharia de Computação

Relatório: Trabalho Prático 2

NP Completo - Partição

Professor: Walisson Ferreira de Carvalho

Arthur Gonçalves Ayres Lanna
Marcus Leandro Gomes Campos Oliveira
Rafael Ramos de Andrade

Belo Horizonte
Campus Coração Eucarístico

3 de dezembro de 2024

Conteúdo

1 Introdução

1.1 Objetivo e escolha do problema

Este trabalho tem como objetivo realizar um estudo comparativo de diferentes abordagens para resolver um problema NP-Completo. A ideia central desse estudo é demonstrar métodos e teorias que tentam resolver um determinado algoritmo de classe NP Completo. Os métodos utilizados foram escolhidos pela sua eficácia e imparcialidade para análise de resultados, com base em casos de teste diversos. As teorias que buscam explicar e resolver problemas dessa categoria, foram retiradas de livros e artigos de autores especializados no assunto.

O problema escolhido para o presente trabalho foi o Problema da Partição. A escolha desse problema se dá a sua premissa simples e interessante. O interesse se deu pela natureza do problema ser comum em aplicações modernas, por se tratar de uma operação simples de comparação de vetores.

1.2 Importância do assunto

Esse trabalho é de grande relevância na área da computação, pois se trata de um assunto que não possui um desfecho. Existem implementações modernas que resolvem esse problema de forma, consideravelmente, eficiente para um NP Completo. Mesmo possuindo essas implementações, o estudo de um algoritmo NP Completo pode colaborar com a resolução de outro NP Completo.

2 Análise de complexidade

2.1 Problema da Partição

O problema da partição (*Partition Problem*) é um dos problemas clássicos em teoria da complexidade computacional e pertence à classe dos problemas NP-completos, como demonstrado por Garey e Johnson [1]. Ele pode ser descrito da seguinte forma: dado um conjunto $S = \{s_1, s_2, \ldots, s_n\}$ de n números inteiros, o objetivo é determinar se é possível dividir S em dois subconjuntos disjuntos S_1 e S_2 de modo que a soma dos elementos de S_1 seja igual à soma dos elementos de S_2, isto é, $\sum_{s \in S_1} s = \sum_{s \in S_2} s$.

A classificação do problema da partição como NP-completo implica que:

1. Ele está em NP, pois uma solução candidata (os subconjuntos S_1 e S_2) pode ser verificada em tempo polinomial.

2. Ele é pelo menos tão difícil quanto qualquer outro problema em NP, uma vez que pode ser reduzido em tempo polinomial a partir de outros problemas NP-completos, como o problema da soma subconjunta (*Subset Sum*) [2].

O problema da partição é relevante em diversas áreas, como otimização, teoria dos números e aplicações práticas, incluindo alocação de recursos e balanceamento de carga em sistemas computacionais [3]. Embora o problema seja NP-completo no caso geral, existem abordagens específicas para resolvê-lo em determinadas circunstâncias:

- **Recursão simples**: A abordagem recursiva explora todas as possíveis partições do conjunto, caracterizando-se por uma complexidade exponencial $O(2^n)$. Apesar de ser conceitualmente simples, essa abordagem não é eficiente, dado o número significativo de subproblemas repetidos, o que a torna impraticável para conjuntos maiores [4].

- **Top-Down com memoization**: Essa técnica melhora a eficiência da recursão utilizando uma tabela (ou cache) para armazenar os resultados dos subproblemas já resolvidos. Dessa forma, evita-se a recalculação redundante, reduzindo a complexidade para $O(n \cdot T)$, onde T é a soma total dos elementos do conjunto. O uso de memoization é especialmente

vantajoso em problemas onde apenas uma parte das combinações possíveis é explorada, como no cálculo de subsequências ou caminhos otimizados [4].

- **Bottom-Up com tabulação**: Neste método, resolve-se os subproblemas menores iterativamente, partindo dos casos base até o problema completo. A tabulação utiliza uma tabela para armazenar os resultados intermediários de forma sistemática, garantindo que cada subproblema seja resolvido uma única vez. Essa abordagem elimina a sobrecarga de chamadas recursivas e o risco de estouro de pilha, com a mesma eficiência de $O(n \cdot T)$. Exemplos incluem algoritmos para subsequências e cálculo de somas parciais [4, 1].
- **Bottom-Up com otimização de espaço**: Esta variação da tabulação reduz o consumo de memória ao utilizar apenas o armazenamento necessário para resolver o subproblema atual. Em vez de manter uma tabela completa, utiliza-se um número fixo de variáveis ou uma estrutura compacta que armazena apenas as últimas iterações relevantes. Essa técnica é eficiente em problemas lineares, como a resolução do problema da mochila ou subsequências [4, 1].

A importância do problema da partição decorre de sua simplicidade e de sua aplicabilidade como modelo para outros problemas de partição e balanceamento. Sua dificuldade intrínseca ressalta as limitações dos algoritmos polinomiais e os desafios relacionados à conjectura $P \neq NP$ [5].

2.2 Versão NP Difícil

A versão NP-hard do problema da partição surge em contextos onde os elementos do conjunto possuem pesos adicionais ou restrições complexas, como no problema da partição com pesos (*Weighted Partition Problem*) ou com múltiplos subconjuntos. Nessa variante, o objetivo é dividir o conjunto $S = \{s_1, s_2, \ldots, s_n\}$ em k subconjuntos disjuntos $S_1, S_2, \ldots, S_k$, tal que as somas dos pesos em cada subconjunto estejam equilibradas, enquanto se respeitam restrições adicionais, como dependências entre os elementos.

Ao contrário da versão clássica, esta generalização é NP-hard, pois aumenta significativamente o espaço de soluções possíveis e incorpora características de outros problemas difíceis, como o problema do caixeiro-viajante (*Traveling Salesman Problem*) e o problema de particionamento de grafos (*Graph Partitioning*) [1, 5].

Essa variante é amplamente utilizada em otimização de sistemas distribuídos, onde o balanceamento de carga entre servidores deve considerar não apenas o volume de trabalho, mas também a conectividade e a latência entre os componentes [3].

3 Metodologia

4 Resultados e análise

Os testes foram realizados em uma máquina com Ryzen 7 5700x e 33 GB de RAM 3200 MHz e os gráficos gerados utilizando Pandas e MatPlotLib.

4.1 Tempo de execução com vetores aleatórios

Os próximo testes foram realizados com vetores aleatórios buscando uma ánalise individual de cada algoritmo, demostrando a tendência de crescimento de cada algoritmo.

Figura 1: Tempo de execução x Tamanho do vetor

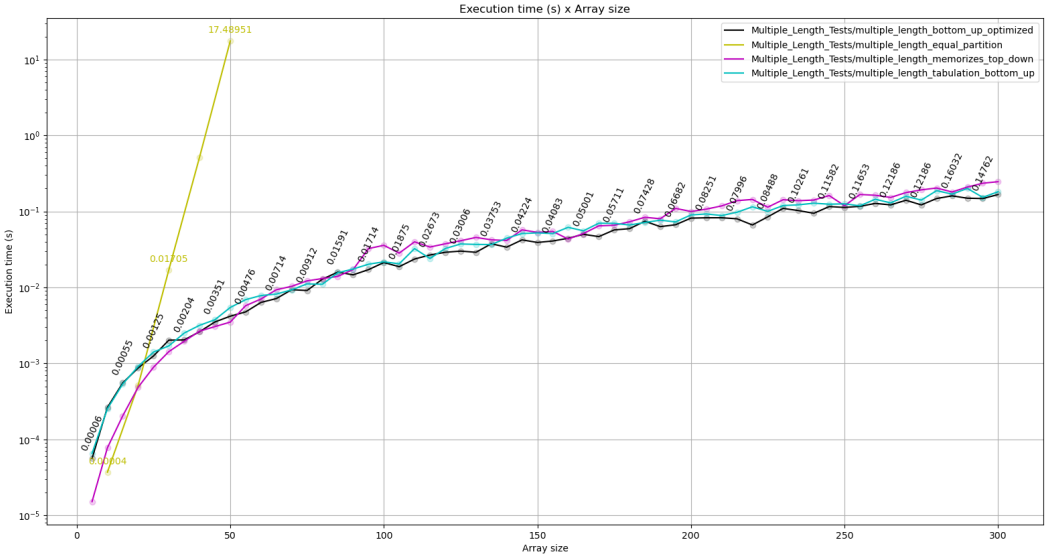


Figura 2: Tempo de execução x Tamanho do vetor

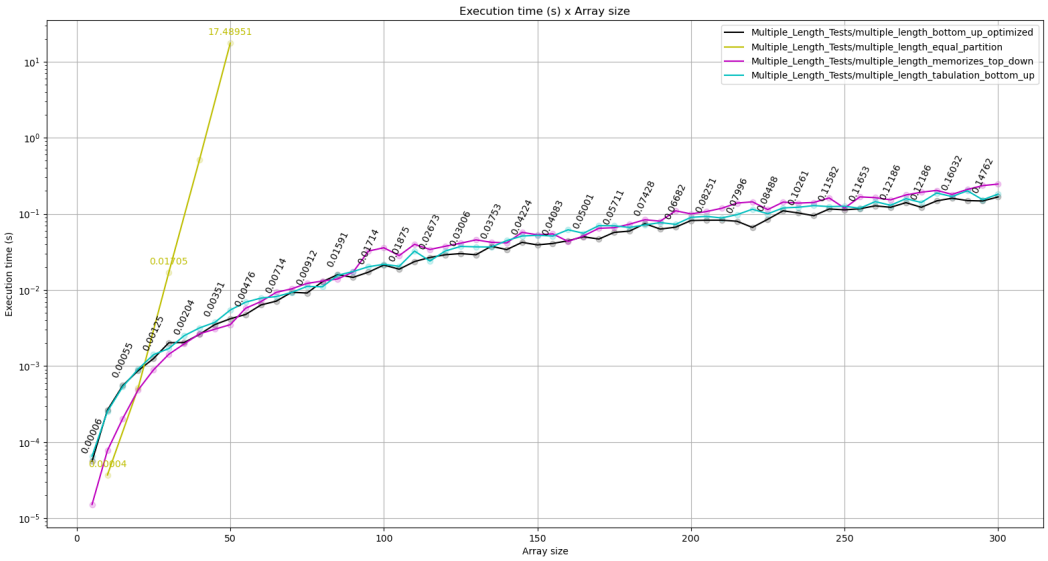


Figura 3: Tempo de execução x Tamanho do vetor

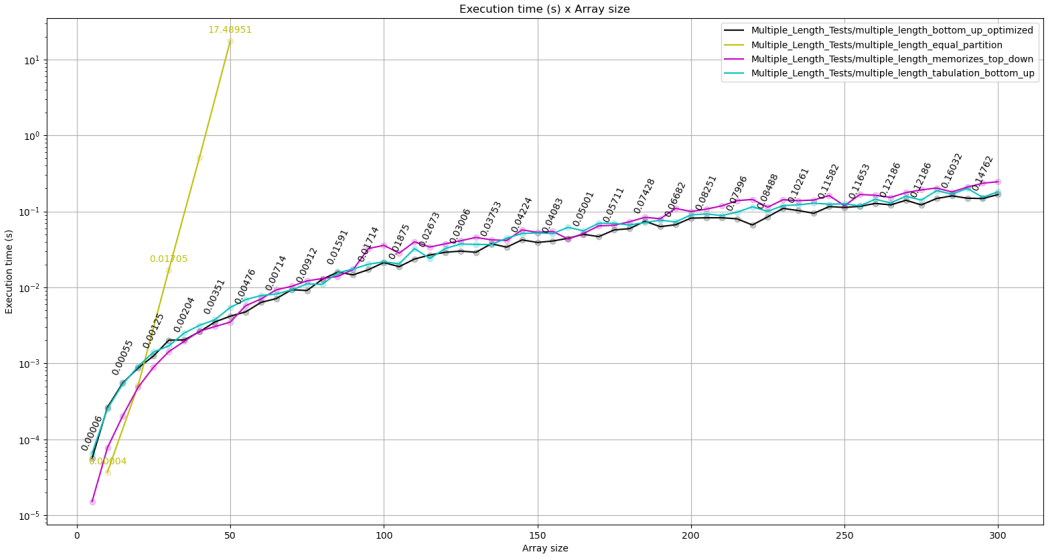
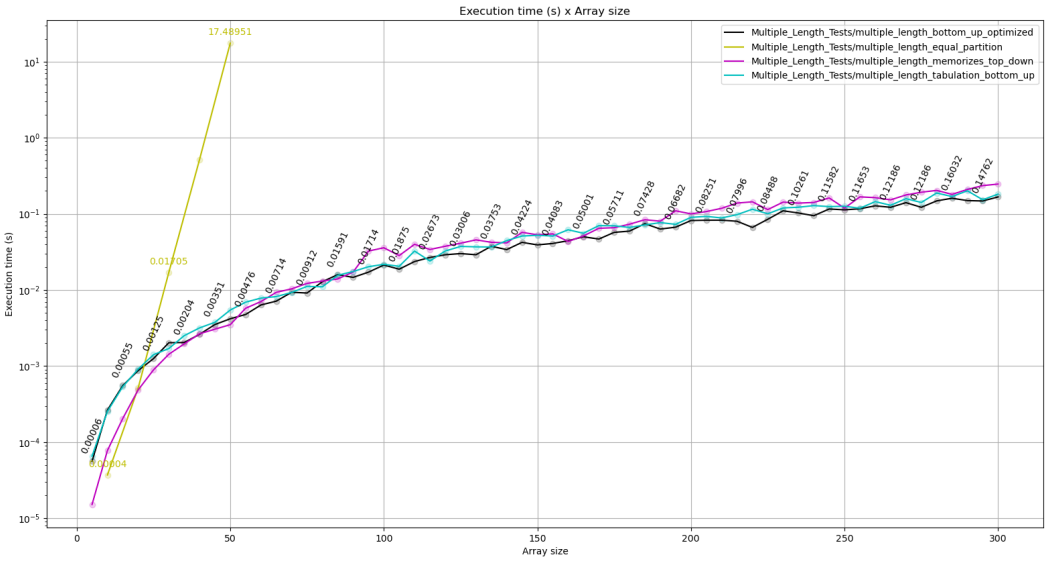


Figura 4: Tempo de execução x Tamanho do vetor



Ao analisarmos os tempos de execução, é possível observar a diferença significativa entre as implementações usando programação dinâmica e otimização de espaço em relação ao uso da força bruta (reta amarela). Para vetores maiores que 22 a implementação por força bruta domina assintoticamente todas as outras implementações. No entanto, para tamanhos menores que 22, ela se mostra extremamente rápida, executando 7 vezes mais rápida que a implementação utilizando programação dinâmica bottom-up, além disso, as implementações utilizando programação dinâmica e otimização de espaço de memória apresentam um desempenho parecido.

4.2 Comparação entre heurística

5 Conclusão

Referências

[1] Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

[2] Karp, R. M. (1972). Reducibility Among Combinatorial Problems. *Complexity of Computer Computations*, 85-103.

[3] Martello, S., & Toth, P. (1981). *Knapsack Problems: Algorithms and Computer Implementations*. Wiley.

[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

[5] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

6 Anexo