Abordagens gulosas e Programação dinâmica para o Problema da mochila

César Tallys, Emerson Souza

Setembro 2024

1 Introdução

O Problema da Mochila é um dos desafios mais clássicos e amplamente estudados na Ciência da Computação e na Otimização Combinatória, sendo referência em pesquisas teóricas e práticas [1]. O problema envolve a escolha de um subconjunto de itens para maximizar o benefício total sem ultrapassar a capacidade de carga de uma mochila. Com aplicações em áreas como alocação eficiente de recursos, planejamento financeiro e logística, o Problema da Mochila reflete questões cruciais em otimização de sistemas. Na sua versão booleana, o Problema da Mochila 0-1, cada item deve ser incluído ou excluído por completo, sem permitir fracionamentos, o que aumenta a complexidade do problema.

Classificado como NP-completo [2], o Problema da Mochila não possui uma solução eficiente que funcione para todas as instâncias em tempo polinomial, especialmente à medida que o número de itens aumenta. Por esse motivo, ele é uma excelente base para a comparação de diferentes abordagens algorítmicas, como algoritmos gulosos e programação dinâmica [3]. Cada uma dessas técnicas oferece benefícios e limitações, como a rapidez na execução dos algoritmos gulosos, que nem sempre garantem a solução ótima, em contraste com a programação dinâmica, que pode ser mais precisa, mas requer maior uso de recursos computacionais. Além disso, este estudo não se limita à análise teórica; também busca realizar um comparativo entre a análise assintótica e a análise empírica dessas abordagens, permitindo uma visão mais abrangente dos desafios e compensações envolvidas na implementação prática dos algoritmos. Assim, serão investigados os principais trade-offs entre a eficiência computacional e a qualidade das soluções obtidas, aspectos cruciais no desenvolvimento de sistemas críticos e de grande escala.

2 Mochila Booleano

O problema da mochila booleano (ou *0-1*) é um problema clássico de otimização combinatória. A versão "booleano" significa que os itens podem ser incluídos ou não na mochila, sem frações. Ou seja, cada item tem duas opções: ser selecionado ou não ser selecionado (daí o nome "0-1").

Definição: Dado um conjunto de n itens, cada item i tem:

- um peso w_i,
- um valor v_i,
- uma capacidade máxima da mochila W.

O objetivo é determinar quais itens incluir na mochila, de forma que:

1. A soma dos pesos dos itens não ultrapasse a capacidade W da mochila.
2. A soma dos valores dos itens seja maximizada.

Formulação matemática: Seja $x_i \in \{0, 1\}$, onde:

- $x_i = 1$ significa que o item i é incluído na mochila,
- $x_i = 0$ significa que o item i não é incluído.

O problema pode ser representado como:

$$\text{Maximizar:} \quad \sum_{i=1}^{n} v_i \cdot x_i$$

$$\text{Sujeito a:} \quad \sum_{i=1}^{n} w_i \cdot x_i \leq W$$

$$x_i \in \{0, 1\}, \quad \text{para todo } i.$$

Existem várias abordagens para resolver o problema da mochila booleano. Uma delas é a **força bruta**, que testa todas as possíveis combinações de itens (2^n combinações), mas essa solução é ineficiente para grandes valores de n. Outra abordagem é a **programação dinâmica**, que resolve o problema em tempo $O(nW)$, sendo mais eficiente para problemas com número de itens moderado e capacidade W controlada. Para instâncias muito grandes, onde a programação dinâmica ainda pode ser inviável, podem ser utilizados algoritmos aproximados, como **algoritmos gulosos**. Este relatório contrasta a aplicação de duas abordagens gulosas e uma abordagem dinâmica, enfatizando seus benefícios em diferentes tamanhos de dados.

2.1 Abordagens Gulosas

2.1.1 Maximização do Número de Itens

Esta abordagem gulosa tenta maximizar a quantidade de itens que podem ser colocados na mochila, escolhendo sempre os itens mais leves primeiro. A estratégia é simples: adicionar itens à mochila até que não haja mais capacidade de peso disponível. O foco aqui é minimizar o peso total, permitindo que o maior número de itens possível seja selecionado.

Algoritmo:

- Os itens são ordenados com base no peso (ordem crescente).
- A cada iteração, o item mais leve é selecionado, desde que o peso total acumulado não exceda a capacidade W da mochila.
- O processo para quando nenhum item adicional pode ser adicionado sem ultrapassar o limite de peso.

Ilustração com exemplo: Imagine que temos uma mochila com capacidade $W = 50$ e os seguintes itens:

- Item 1: valor = 60, peso = 10
- Item 2: valor = 100, peso = 20
- Item 3: valor = 120, peso = 30

Ordenando os itens pelo peso, o algoritmo primeiro adicionaria o Item 1 (peso 10), depois o Item 2 (peso 20), e finalmente o Item 3 (peso 30), acumulando 60 unidades de peso, que excedem a capacidade $W = 50$, então o Item 3 seria ignorado. O foco está em maximizar a quantidade de itens, sem levar em consideração diretamente o valor de cada item.

Limitação: Essa abordagem não garante a solução ótima em termos de valor total dos itens, pois não leva em consideração a relação valor/peso, apenas o peso.

2.1.2 Maximização da Relação Valor/Peso

Descrição Algorítmica: Essa é outra abordagem gulosa, mas, em vez de focar apenas no peso, ela prioriza a melhor relação valor/peso para maximizar o valor total da mochila. O algoritmo calcula a relação valor/peso de cada item e seleciona aqueles com a maior relação, maximizando o valor total.

Algoritmo:

- Para cada item, é calculada a relação valor/peso.
- Os itens são ordenados de acordo com essa relação, de forma decrescente.
- A cada iteração, o item com a melhor relação valor/peso é adicionado à mochila, desde que a capacidade não seja excedida.

Ilustração com exemplo: Utilizando os mesmos itens:

- Item 1: valor = 60, peso = 10 $\Rightarrow 60/10 = 6$
- Item 2: valor = 100, peso = 20 $\Rightarrow 100/20 = 5$
- Item 3: valor = 120, peso = 30 $\Rightarrow 120/30 = 4$

O algoritmo selecionaria primeiro o Item 1, depois o Item 2, e então verificaria que não é possível adicionar o Item 3 sem exceder a capacidade. Aqui, o valor total dos itens na mochila é maximizado, com o foco na eficiência de cada unidade de peso.

Limitação: Embora essa abordagem seja mais eficiente em termos de valor do que a abordagem puramente baseada no peso, ela ainda pode falhar em encontrar a solução ótima em certos cenários onde a combinação de itens menores oferece um valor total melhor.

2.2 Programação Dinâmica

Esta abordagem é uma técnica mais avançada que resolve o problema dividindo-o em subproblemas menores. Ao contrário das abordagens gulosas, a programação dinâmica garante a solução ótima ao considerar todas as combinações possíveis de itens. O algoritmo cria uma tabela onde cada célula representa o valor máximo que pode ser obtido com um determinado conjunto de itens e uma capacidade de mochila específica.

Algoritmo:

- Uma matriz dp é criada, onde $dp[i][w]$ representa o valor máximo que pode ser obtido usando os primeiros i itens e uma capacidade de w na mochila.
- Para cada item, o algoritmo decide se vale a pena adicioná-lo à mochila ou não, comparando os valores resultantes de incluí-lo ou excluí-lo.
- A solução final é encontrada na célula $dp[n][W]$, onde n é o número total de itens e W é a capacidade máxima da mochila.

Ilustração com exemplo: Considerando os mesmos itens e uma mochila com capacidade $W = 50$, o algoritmo constrói a seguinte matriz para determinar a melhor combinação de itens:

- $dp[0][w]$: sem itens, o valor é sempre 0.
- $dp[1][w]$: o Item 1 é considerado, então o valor para capacidades acima de 10 é 60.
- $dp[2][w]$: o Item 2 é considerado, e assim por diante.

Após preencher toda a matriz, o algoritmo recupera a solução ótima, que pode envolver a combinação de itens que oferece o maior valor total dentro da capacidade da mochila.

Limitação: Embora essa abordagem garanta uma solução perfeita, o custo computacional é mais elevado, podendo limitar o tamanho da entrada a depender do hardware utilizado.

3 Análise Assintótica dos Algoritmos

A análise assintótica é uma técnica utilizada para descrever o comportamento de algoritmos à medida que o tamanho de suas entradas cresce indefinidamente. Focada em estimar o tempo de execução ou o uso de recursos, essa abordagem expressa o crescimento em termos de funções matemáticas simplificadas, permitindo a comparação de diferentes algoritmos de forma abstrata. A notação mais utilizada nessa análise é a Big-O, que define o limite superior do desempenho no pior caso.

A seguir, a análise será realizada considerando as três abordagens destacadas ao longo do artigo, avaliando o desempenho do algoritmo.

3.1 Abordagem Gulosa (Peso)

Essa abordagem tenta maximizar o número de itens na mochila, ordenando-os pelo peso e adicionando o maior número possível de itens sem exceder o limite de peso.

Passo 1: Ordenação dos Itens

O que acontece: O algoritmo precisa ordenar os itens com base no peso. Isso foi feito usando o método `sort` do JavaScript, que é uma combinação de Merge Sort e Insertion Sort, com uma complexidade de tempo de $O(n \log n)$, onde n é o número de itens.
Tempo de execução: Para um número de itens n, a ordenação por peso leva $O(n \log n)$. Esse é o passo que consome mais tempo no algoritmo. Quanto maior o número de itens, mais tempo leva para ordená-los. Para $n = 1000$, a ordenação pode levar entre 1 e 10 milissegundos, dependendo da implementação e do hardware utilizado.

Passo 2: Seleção dos Itens

O que acontece: Após a ordenação, o algoritmo percorre a lista de itens e vai adicionando-os na mochila, verificando se o peso total não ultrapassa a capacidade W.
Tempo de execução: Essa operação é realizada uma vez para cada item, o que resulta em uma complexidade de tempo linear, ou seja, $O(n)$. Nesse caso, o tempo gasto para essa parte é proporcional ao número de itens. Para $n = 1000$, a iteração por todos os itens pode levar algo em torno de 1 a 5 milissegundos.

Passo 3: Construção do Vetor de Seleção

O que acontece: Para registrar quais itens foram selecionados, o algoritmo mantém um vetor de seleção de tamanho n, onde cada posição do vetor representa se o item correspondente foi escolhido ou não.
Tempo de execução: Preencher o vetor de seleção também tem complexidade linear $O(n)$, pois cada item será marcado uma vez. Isso geralmente consome menos tempo que os dois passos anteriores.

Tempo Total do Algoritmo

O tempo total de execução é dominado pela etapa de ordenação, portanto, o tempo total é $O(n \log n)$. A etapa de seleção e a construção do vetor de seleção são secundárias e têm tempo $O(n)$, mas o tempo final é, no geral, $O(n \log n)$.

3.2 Abordagem Gulosa (Valor/Peso)

Essa abordagem tenta maximizar o valor da mochila, priorizando a melhor relação entre valor e peso $(\frac{\text{valor}}{\text{peso}})$.

Passo 1: Cálculo da Relação Valor/Peso

O que acontece: O algoritmo precisa calcular a relação valor/peso para cada item. Isso é feito uma vez para cada item.
Tempo de execução: Essa operação tem complexidade $O(n)$, pois o cálculo de $\frac{\text{valor}}{\text{peso}}$ para cada item é uma operação constante. Para $n = 1000$, calcular a relação valor/peso de todos os itens pode levar cerca de 1 milissegundo.

Passo 2: Ordenação dos Itens

O que acontece: Os itens são ordenados de acordo com sua relação valor/peso. A ordenação também tem complexidade de tempo $O(n \log n)$.
Tempo de execução: Assim como na abordagem gulosa baseada no peso, o tempo de ordenação dos itens domina o tempo de execução. Para $n = 1000$, esse passo pode levar entre 1 e 10 milissegundos.

Passo 3: Seleção dos Itens

O que acontece: O algoritmo percorre a lista de itens já ordenada e seleciona os itens até que o peso total atinja o limite W. Isso é feito uma vez para cada item.
Tempo de execução: A seleção tem complexidade linear, $O(n)$, e segue o mesmo padrão da abordagem anterior. Para $n = 1000$, esse passo pode levar em torno de 1 a 5 milissegundos.

Passo 4: Construção do Vetor de Seleção

O que acontece: Assim como na abordagem anterior, o algoritmo mantém um vetor de seleção para registrar quais itens foram escolhidos.
Tempo de execução: Preencher o vetor de seleção também tem complexidade linear $O(n)$.

Tempo Total do Algoritmo

O tempo total deste algoritmo também é dominado pela ordenação dos itens, com tempo $O(n \log n)$. A etapa de cálculo da relação valor/peso e a etapa de seleção têm complexidade $O(n)$, mas o tempo total ainda é $O(n \log n)$, similar à abordagem baseada apenas no peso.

3.3 Programação Dinâmica

A programação dinâmica resolve o problema da mochila de maneira exata, utilizando uma tabela para armazenar soluções parciais.

Passo 1: Inicialização da Tabela

O que acontece: O algoritmo inicia uma tabela $dp[n+1][W+1]$, onde cada célula representa o valor máximo que pode ser obtido com os primeiros i itens e uma capacidade de mochila de W.
Tempo de execução: A inicialização desta tabela tem complexidade $O(n \cdot W)$, pois todas as células precisam ser criadas e inicializadas com valores padrão. Se $n = 1000$ e $W = 1000$, essa inicialização pode levar cerca de 5 a 20 milissegundos, dependendo da implementação.

Passo 2: Preenchimento da Tabela

O que acontece: O algoritmo percorre cada célula da tabela e preenche com base em duas opções: ou o item i é incluído na solução ou não. Isso é feito em tempo constante para cada célula da tabela, e como a tabela tem $n \times W$ células, o tempo total para preenchê-la é $O(n \cdot W)$.
Tempo de execução: Esta é a parte mais custosa do algoritmo. Para grandes valores de W, o tempo

pode aumentar drasticamente. Por exemplo, para $n = 1000$ e $W = 1000$, o tempo pode levar entre 100 a 500 milissegundos.

Passo 3: Recuperação dos Itens Selecionados

O que acontece: Após preencher a tabela, o algoritmo reconstrói a solução, retrocedendo pelas células da tabela para determinar quais itens foram incluídos na mochila.
Tempo de execução: A recuperação dos itens selecionados tem complexidade linear $O(n)$, já que o algoritmo percorre os itens uma vez para determinar quais foram selecionados.

Tempo Total do Algoritmo

O tempo total para este algoritmo é dominado pela fase de preenchimento da tabela, com complexidade $O(n \cdot W)$. Esta abordagem pode ser significativamente mais lenta que as abordagens gulosas quando W é grande, especialmente para grandes instâncias. A fase de inicialização e a recuperação dos itens adicionam $O(n)$ ao tempo total, mas o tempo total continua sendo $O(n \cdot W)$.

4 Materiais e métodos

4.1 Materiais Utilizados

Para a realização dos experimentos de comparação entre execuções das abordagens algorítmicas para o Problema da Mochila Booleano, foram utilizados os seguintes materiais:

4.1.1 Hardware

- **Processador:** AMD Ryzen 5 4600G
- **Memória:** 16 GB a 3200 MHz

4.1.2 Software

- **Node.js**: Ambiente de execução de JavaScript no lado do servidor, versão 20.16.0.
- **Visual Studio Code (VSCode)**: Editor de texto e IDE para desenvolvimento de código.
- **Terminal integrado (PowerShell)**: Para execução de comandos diretamente no ambiente de desenvolvimento.

4.2 Método para Execução dos experimentos

A seguir, descreve-se o passo a passo utilizado para executar os experimentos e comparar as diferentes abordagens para o Problema da Mochila Booleano.

4.2.1 Configuração do ambiente

- **Instalação do Node.js**: O ambiente de execução de JavaScript foi instalado no sistema para que os algoritmos pudessem ser executados diretamente no terminal. A versão utilizada foi a 20.16.0, que pode ser obtida no *Node.js official website*.
- **Criação dos Arquivos de Código**: Clone o projeto diretamente do nosso repositório no github[4].

4.2.2 Execução dos algoritmos

Cada um dos algoritmos foi executado de forma independente no terminal, utilizando o comando `node` para rodar o código JavaScript. O processo foi o seguinte:

- **Navegação até o diretório**: Dentro da raiz do projeto clonado, abra o terminal e execute cada um dos algoritmos conforme os passos seguinte:
- **Execução do algoritmo 1 (Maximização de Itens)**: O primeiro algoritmo, que utiliza uma abordagem gulosa baseada no peso dos itens, foi executado com o seguinte comando:

  ```
  node 1-maxItems.js
  ```

  O script lê os dados do arquivo de instância, processa os itens e retorna o número de itens selecionados, o peso total e o vetor de seleção. O resultado foi armazenado para posterior análise.
- **Execução do algoritmo 2 (Maximização da Relação Valor/Peso)**: O segundo algoritmo, que utiliza a estratégia de relação valor/peso para maximizar o valor total, foi executado da mesma forma:

  ```
  node 2-maxBenefit.js
  ```

  Da mesma maneira, os resultados retornados pelo algoritmo foram gravados.
- **Execução do algoritmo 3 (Programação Dinâmica)**: Finalmente, o algoritmo de programação dinâmica, que fornece a solução ótima para o problema, foi executado com o comando:

  ```
  node 3-dynamic.js
  ```

  Se necessário disponibilizar mais memória para a execução:

  ```
  node --max-old-space-size=4096 3-dynamic.js
  ```

4.2.3 Coleta de resultados

Após a execução de cada algoritmo, os seguintes dados foram coletados e registrados:

- Número total de itens selecionados.
- Peso total dos itens incluídos na mochila.
- Valor total dos itens.
- Vetor de seleção, indicando quais itens foram escolhidos (1 para itens selecionados, 0 para itens não selecionados).

Os resultados foram organizados em tabelas para comparação, focando no tempo de execução, eficiência de valor/peso e a solução ótima obtida com a programação dinâmica.

4.2.4 Análise de Desempenho

Para medir o desempenho dos algoritmos em termos de tempo de execução, foi utilizada a função `performance.now() (módulo nativo do Node.js a partir da versão 8.5)` em cada arquivo JavaScript para medir o tempo gasto para processar os dados e obter o resultado.

Isso permitiu comparar a eficiência de cada abordagem em cenários com diferentes números de itens e capacidades de mochila.

5 Análise empírica

Para realizar a análise de desempenho dos três algoritmos utilizados para resolver o Problema da Mochila Booleano, foi necessário medir o tempo de execução, o número de itens selecionados, o peso total e o valor total, determinando a eficiência de cada abordagem em termos de tempo e qualidade da solução.

Os dados foram capturados para cada um dos métodos, usando 4 tamanhos de entradas para cada uma duas pastas[4] contendo Arrays de baixa e alta escala. Para uma melhor visualização dos resultados, os dados serão apresentados em forma de tabela, onde n representa a quantidade de itens disponíveis e W a capacidade de peso da mochila.

5.1 Entradas de baixa dimensão

Table 1: Comparativo de Eficiência: Quantidade de Itens (n) e Peso Total (w)

Algoritmo	**Valores de N e W**			
	4, 20	**10, 269**	**20, 879**	**23, 10000**
Max. Itens	3, 18	6, 227	17, 818	13, 9653
Max. Benefício	3, 18	6, 260	17, 837	11, 9750
Prog. Dinâmica	3, 18	6, 269	17, 871	11, 9768

Table 2: Comparativo de Tempos de Execução (em milissegundos)

Algoritmo	**Valores de N e W**			
	4, 20	**10, 269**	**20, 879**	**23, 10000**
Max. Itens	0,128 ms	0,225 ms	0,260 ms	0,343 ms
Max. Benefício	0,158 ms	0,249 ms	0,273 ms	0,323 ms
Prog. Dinâmica	0,324 ms	0,616 ms	2,397 ms	12,248 ms

Ao analisar os resultados dos três algoritmos aplicados a entradas de baixa escala (N variando de 4 a 23), podemos observar o seguinte comportamento:

5.1.1 Maximização por Itens

- **Eficiência**: O algoritmo consegue inserir a maior quantidade de itens nos casos testados, com uma eficiência boa em relação ao peso total (o peso quase sempre se aproxima do limite da mochila, exceto no último caso).

- **Tempo de Execução**: O tempo de execução é extremamente rápido, com execuções variando entre 0,128 ms e 0,343 ms. Isso mostra que o algoritmo é muito eficiente para pequenas entradas, visto que o tempo é praticamente insignificante.
- **Análise Geral**: Para entradas pequenas, essa abordagem oferece bons resultados em termos de quantidade de itens e peso total, além de manter o tempo de execução em níveis muito baixos. Ela pode ser considerada uma solução viável quando simplicidade e velocidade são prioritárias.

5.1.2 Maximização por Benefício

- **Eficiência**: O algoritmo faz uma melhor escolha de itens em termos de benefício/custo, o que pode ser observado pelo fato de que, em alguns casos, ele obtém um peso total maior do que o algoritmo que maximiza itens (por exemplo, N=10 e N=20). No entanto, o número de itens inseridos é menor (especialmente em N=23), o que indica uma preferência por itens mais valiosos em termos de benefício.
- **Tempo de Execução**: O tempo de execução também é muito rápido, variando entre 0,158 ms e 0,323 ms. Embora um pouco mais lento que a maximização por itens, a diferença é praticamente insignificante para entradas pequenas.
- **Análise Geral**: Para entradas de baixa escala, essa abordagem oferece uma boa otimização em termos de benefício, sem comprometer significativamente o tempo de execução. É uma boa escolha quando a qualidade (benefício) dos itens é mais importante do que a quantidade.

5.1.3 Programação Dinâmica

- **Eficiência**: A programação dinâmica se comporta de maneira similar ao algoritmo de maximização por benefício, geralmente alcançando o peso total máximo. No entanto, o número de itens escolhidos é o menor entre os três algoritmos, sugerindo que a escolha de itens é muito criteriosa para alcançar o valor ideal.
- **Tempo de Execução**: Apesar de sua eficiência em termos de minimização do peso, o custo em tempo de execução é notável. O tempo de execução, embora ainda baixo (entre 0,324 ms e 12,248 ms), é significativamente maior que os outros dois algoritmos. Isso se torna mais evidente em N=23, onde o tempo de execução sobe para 12,248 ms, indicando que, mesmo em escalas pequenas, a programação dinâmica pode ser mais lenta.
- **Análise Geral**: A programação dinâmica é eficiente em termos de solução ótima, e em entradas pequenas, apesar de ser matematicamente mais cara, ela ainda é recomendada por dar a solução ótima com um tempo de execução extremamente baixo.

5.1.4 Conclusões

- **Maximização por Itens** é extremamente eficiente em termos de tempo para entradas pequenas, mas pode não ser ideal se a otimização de benefício for o objetivo.
- **Maximização por Benefício** oferece um bom equilíbrio entre eficiência de tempo e qualidade das soluções (peso total e número de itens), sendo uma ótima escolha em casos onde a maximização do benefício é o foco.
- **Programação Dinâmica** garante uma solução ótima. Em entradas pequenas o seu tempo de execução, embora maior que os outros, ainda é muito baixo, tornando-a a melhor opção.

Portanto, em entradas de baixa escala, a **programação dinâmica** é mais eficiente, ficando a critério de gerente de projeto, escolher se prefere uma solução perfeita ou ganhar alguns milésimos de segundo em tempo de execução.

5.2 Entradas de larga escala

Table 3: Comparativo de Eficiência: Quantidade de Itens (n) e Peso Total (w)

Algoritmo	**Valores de N e W**			
	100, 995	**500, 2543**	**2000, 10011**	**10000, 49877**
Max. Itens	13, 965	47, 2514	181, 9943	955, 49785
Max. Benefício	12, 908	43, 2528	162, 9996	841, 49876
Prog. Dinâmica	12, 985	42, 2543	160, 10011	840, 49877

Table 4: Comparativo de Tempos de Execução (em milissegundos)

Algoritmo	**Valores de N e W**			
	100, 995	**500, 2543**	**2000, 10011**	**10000, 49877**
Max. Itens	0,188 ms	0,385 ms	1,375 ms	5,028 ms
Max. Benefício	0,231 ms	0,512 ms	3,337 ms	6,837 ms
Prog. Dinâmica	6,146 ms	24,599 ms	338,380 ms	7.323,704 ms

Para as entradas de larga escala (N variando de 100 a 10000), os resultados obtidos destacam diferenças mais pronunciadas no comportamento dos três algoritmos. A análise abaixo considera tanto a eficiência na escolha dos itens quanto o tempo de execução.

5.2.1 Comparativo de Eficiência: Quantidade de Itens e Peso Total

- Para entradas de larga escala, a quantidade de itens n e o peso total w mostram que, de modo geral, o algoritmo de **maximização por itens** insere mais itens em relação aos outros, mantendo o peso total próximo ao limite da capacidade da mochila em todos os casos testados.
- No entanto, o algoritmo de **maximização por benefício**, embora selecione menos itens, tende a encontrar uma solução com um peso total muito próximo do limite W, especialmente nos casos de $N = 500$ e $N = 2000$. Isso indica que o algoritmo está focado na escolha de itens que maximizam o benefício, o que pode resultar na inserção de itens mais pesados.
- A **programação dinâmica** apresenta um comportamento semelhante ao de maximização por benefício, atingindo sempre o peso total mais próximo ao limite W, mas com o menor número de itens escolhidos entre os três algoritmos. Isso sugere uma maior eficiência em termos de valor da solução, mesmo que menos itens sejam escolhidos.

5.2.2 Comparativo de Tempos de Execução

- O tempo de execução é o fator mais significativo ao comparar os três algoritmos em entradas de larga escala. A **maximização por itens** apresenta um desempenho significativamente mais rápido, com tempos de execução variando de 0,188 ms a 5,028 ms. O aumento no tempo é linear com o aumento de N, indicando uma escalabilidade muito boa.
- A **maximização por benefício**, embora também apresente tempos de execução baixos em comparação com a programação dinâmica, tem um desempenho ligeiramente mais lento que a maximização por itens, com tempos variando de 0,231 ms a 6,837 ms. O aumento do tempo de execução também é linear, mas o tempo é proporcionalmente maior devido à maior complexidade envolvida na escolha dos itens com maior benefício.

- A **programação dinâmica**, por outro lado, apresenta uma disparada significativa no tempo de execução à medida que N aumenta. Com tempos que variam de 6,146 ms a impressionantes 7.323,704 ms, o algoritmo claramente não escala bem para grandes entradas, mostrando um crescimento exponencial no tempo de execução.

6 Discussão Geral: Comparação entre Entradas de Baixa e Larga Escala

A análise dos resultados revela um comportamento distinto dos algoritmos ao lidar com entradas de baixa e larga escala.

6.1 Escalabilidade dos Algoritmos

- Para **entradas de baixa escala**, todos os algoritmos apresentaram tempos de execução extremamente baixos, tornando a escolha entre eles menos significativa em termos de desempenho. No entanto, à medida que as entradas aumentam, essa diferença se torna muito mais pronunciada.

- O **algoritmo de maximização por itens** demonstrou a melhor escalabilidade, com um crescimento quase linear no tempo de execução, tanto para entradas pequenas quanto para entradas grandes. Isso o torna uma escolha viável para problemas de larga escala onde a simplicidade e a velocidade de execução são essenciais.

- O **algoritmo de maximização por benefício** também se mostrou eficiente em termos de escalabilidade, apresentando um crescimento linear no tempo de execução. No entanto, a diferença de tempo em relação à maximização por itens é mais pronunciada em entradas de larga escala, o que pode impactar sua viabilidade quando o desempenho é uma prioridade.

- A **programação dinâmica**, embora ofereça uma solução ótima, mostrou-se inadequada para problemas de larga escala devido ao crescimento exponencial no tempo de execução. Esse comportamento foi observado de maneira muito evidente em $N = 10000$, onde o tempo de execução atingiu mais de 7 segundos, um valor extremamente alto em comparação com os outros dois algoritmos.

6.2 Qualidade da Solução

- Em termos de qualidade da solução, a **maximização por itens** oferece uma maior quantidade de itens e uma proximidade considerável com o peso total W, pode não ser a melhor escolha se o objetivo for maximizar o benefício obtido.

- A **maximização por benefício** oferece uma solução mais eficiente em termos de valor obtido, mesmo que insira menos itens na mochila. Esse comportamento se torna mais relevante em problemas de larga escala, onde a escolha dos itens certos pode fazer uma grande diferença no benefício total.

- A **programação dinâmica**, embora seja mais lenta, oferece uma solução ótima que reduz a quantidade de itens, enquanto maximiza o benefício. No entanto, o custo computacional dessa solução pode torná-la inviável para problemas de larga escala, onde o tempo de execução é uma preocupação importante.

6.3 Conclusão

Os resultados mostram que, para entradas de baixa escala, todos os algoritmos são viáveis em termos de tempo de execução, e a escolha pode ser baseada na qualidade da solução desejada. No entanto,

para entradas de larga escala, a maximização por itens se destaca como a solução mais escalável, enquanto a programação dinâmica, embora ótima em termos de qualidade da solução, não é adequada devido ao seu alto custo tanto em tempo de execução quanto em consumo de memória.

Consumo de Memória: A programação dinâmica utiliza uma tabela bidimensional para armazenar os subproblemas intermediários, com dimensões baseadas em N e W. Esse método consome uma quantidade significativa de memória, especialmente em problemas de larga escala, onde W pode ser um número grande. O uso dessa tabela permite a obtenção da solução ótima, mas torna o algoritmo menos eficiente do ponto de vista de recursos computacionais. Para grandes valores de N e W, o espaço de memória necessário aumenta exponencialmente, resultando em um consumo de memória que pode ser proibitivo em sistemas com recursos limitados.

A maximização por itens e a maximização por benefício, por outro lado, têm um consumo de memória significativamente menor, pois não precisam manter tabelas intermediárias para cada subproblema. Esses algoritmos, portanto, são mais adequados para entradas de larga escala, onde a eficiência de memória e tempo de execução são cruciais.

6.4 Recomendações

Dado o desempenho de cada algoritmo, as recomendações são as seguintes:

- Para entradas de larga escala, onde o tempo de execução e o consumo de memória são fatores críticos, a **maximização por itens** é a melhor escolha, oferecendo uma solução rápida e eficiente.
- Se a qualidade da solução é mais importante que o tempo de execução, mas o consumo de memória deve ser controlado, a **maximização por benefício** oferece um bom equilíbrio entre as duas dimensões.
- A **programação dinâmica**, embora produza a solução ótima, só deve ser utilizada em casos onde o tempo e a memória não são limitações severas, ou em problemas de menor escala, onde o custo computacional ainda é administrável.

7 Considerações Finais

O objetivo principal deste trabalho foi realizar uma comparação entre a análise empírica e assintótica dos três algoritmos aplicados ao Problema da Mochila Booleano: maximização por itens, maximização por benefício e programação dinâmica. A partir dos resultados obtidos, observou-se que cada algoritmo apresenta comportamentos distintos, dependendo da escala da entrada (pequena ou grande) e das métricas consideradas (tempo de execução, qualidade da solução e consumo de memória).

Em termos de **tempo de execução**, os algoritmos de maximização por itens e maximização por benefício mostraram um comportamento linear, com tempos de execução muito rápidos, especialmente para entradas de grande escala. Já a programação dinâmica, apesar de garantir a solução ótima, teve um crescimento exponencial no tempo de execução, o que a torna inviável para problemas de grande porte.

No que se refere à **qualidade da solução**, a programação dinâmica, como esperado, apresentou os melhores resultados, maximizando tanto o benefício quanto o peso total permitido pela mochila. Entretanto, os algoritmos de maximização por itens e maximização por benefício também foram eficientes, principalmente em termos de aproximação do valor ótimo, com um tempo de execução muito menor.

Quanto ao **consumo de memória**, a programação dinâmica, com seu uso intensivo de tabelas para armazenar subproblemas intermediários, mostrou-se significativamente mais custosa, especialmente para entradas de grande escala. Em contrapartida, os algoritmos de maximização por itens e por benefício apresentaram um consumo de memória mais eficiente, tornando-os mais adequados para problemas que envolvem grandes quantidades de dados.

Diante disso, a escolha do algoritmo mais adequado depende do cenário específico. Para entradas pequenas, todos os algoritmos são viáveis, e a decisão pode ser baseada na qualidade da solução desejada. Para entradas maiores, no entanto, a maximização por itens se destaca pela excelente escalabilidade, sendo a escolha mais adequada quando o tempo de execução e o consumo de memória são prioridades. Quando a qualidade da solução é o foco, mas o desempenho ainda é um fator relevante, a maximização por benefício apresenta um bom equilíbrio entre eficiência e benefício. A programação dinâmica, embora ótima em termos de qualidade da solução, deve ser usada com cautela devido ao seu alto custo em termos de tempo e memória, sendo recomendada apenas em cenários onde esses fatores não são limitantes.

Este estudo reforça a importância de uma análise criteriosa ao escolher algoritmos de resolução de problemas combinatórios, considerando não apenas a complexidade teórica, mas também o comportamento prático em diferentes escalas de entrada.

8 Referências

1. Martello S., Toth P. (1990). Knapsack Problems: Algorithms and Computer Implementations. John Wiley Sons.

2. Karp, R. M. (1972). Reducibility among combinatorial problems. In Complexity of Computer Computations (pp. 85-103). Springer.

3. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. (2009). Introduction to Algorithms. MIT Press.

4. Github. Repositório dos algoritmos.
`https://github.com/CesarTHD/Algoritmos-gulosos-e-programacao-dinamica`