Análise e Revisão do Problema da Mochila 0-1

Alberto Magno Machado[1], **Josué Pereira Nogueira**[1], **Leonardo Henrique Saraiva de Avelar**[1]

[1]Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Projeto e Análise de Algoritmos – Prof. Walisson Ferreira de Carvalho

Abstract. *The 0-1 knapsack problem is one of the most classic and studied problems in combinatorial optimization. This paper aims to provide an introductory and review analysis of this problem, addressing its mathematical formulation, computational complexity, variants, and both exact and approximate solution techniques. The discussion is grounded in classical works and recent scientific literature.*

Resumo. *Este artigo apresenta uma análise introdutória e revisional do problema da mochila 0-1, um problema clássico da otimização combinatória e exemplo canônico de problema NP-completo. São abordados sua formulação matemática, a complexidade computacional, as variantes do problema e os principais métodos de solução (exatos e aproximados), com base na literatura especializada.*

1. Introdução

O problema da mochila 0-1 é um clássico problema de otimização combinatória em que se deve selecionar, a partir de um conjunto finito de itens, aqueles que serão carregados em uma "mochila" com capacidade limitada, de modo a maximizar o valor total transportado. Cada item i possui um peso w_i e um valor v_i, e não é permitido fracionar ou repetir itens – cada item é escolhido ou não (daí o "0-1").

> Dado n itens, onde cada item possui um peso e um lucro associados, e também dado uma mochila com capacidade W (ou seja, a mochila pode conter no máximo W de peso), a tarefa é colocar os itens na mochila de modo que a soma dos lucros associados seja a máxima possível. A restrição é que só podemos colocar um item completamente na mochila ou não colocá-lo de forma alguma (não é possível colocar apenas uma parte de um item na mochila).

Fonte: tradução livre de GeeksforGeeks [GeeksforGeeks 2025].

Formalmente, o objetivo é maximizar o valor total $\sum_{i=1}^{n} v_i x_i$ sujeito à restrição de que o peso total $\sum_{i=1}^{n} w_i x_i$ não exceda a capacidade da mochila W, com $x_i \in \{0, 1\}$ indicando a inclusão do item.

Ou seja:

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{sujeito a} \quad \sum_{i=1}^{n} w_i x_i \leq W, \quad x_i \in \{0, 1\}.$$

Isso é, dado um conjunto finito de n itens, cada um com peso w_i e valor v_i, deseja-se escolher um subconjunto de itens de modo que seja maximizado o valor total transportado, respeitando a capacidade da mochila W, tendo sempre duas possibilidades, inserir o item na mochila ($x_i = 1$) ou não ($x_i = 0$).

Portanto temos um problema de decisão (cuja resposta é sim ou não) sendo esse um problema NP-completo. A versão de decisão — "existe um subconjunto com valor total pelo menos V?" — é NP-completa [Garey and Johnson 1979, Karp 1972].

Problemas da classe NP são não determinísticos em tempo polinomial, problemas NP-Completos como o da mochila 0-1 são problemas de decisão cuja solução pode ser verificada em tempo polinomial, mas que não se conhece um algoritmo eficiente para resolvê-los em tempo polinomial, além disso, eles são redutíveis entre si, ou seja, se descobrirem uma forma de resolver um NP-completo em tempo polinomial, todos os outros problemas NP-completos também podem ser resolvidos em tempo polinomial.

O problema da mochila 0-1 é um dos problemas mais clássicos e estudados na otimização combinatória, sendo um exemplo canônico de problema NP-completo. Seu nome deriva da metáfora de um viajante que deve otimizar a carga de uma mochila limitada, buscando levar o máximo de valor possível. O problema apresenta grande relevância teórica e prática, sendo estudado há mais de um século e aparecendo em diversas aplicações reais, como corte de materiais, seleção de portfólios e sistemas de criptografia.

Se originou de estudos de otimização do início do século XX. A formulação combinatória que conhecemos hoje foi amplamente divulgada com o desenvolvimento de métodos de programação dinâmica por Bellman na década de 1950 [Nemhauser and Wolsey 1988], e ganhou grande atenção teórica quando sua forma de decisão foi provada NP-completa por Karp em 1972 [Karp 1972], dando sequência às definições de problemas intratáveis consolidadas em [Garey and Johnson 1979]. Desde então, o problema tem sido um estudo de referência em algoritmos exatos, heurísticos e esquemas de aproximação (FPTAS) [Ibarra and Kim 1975, Jin 2019].

Casos reais em produção incluem:

- **Ant Financial (Alibaba)**: desenvolveu um solucionador distribuido capaz de resolver instâncias com 1 bilhão de variáveis e restrições em cerca de uma hora, utilizado diariamente em produção, com impacto significativo em alocação de recursos e marketing (Knapsack solver via Hadoop/Spark/MPI) [Zhang et al. 2020].
- **Amazon PackOpt**: implementou o projeto PackOpt, criando algoritmos para selecionar caixas ideais que reduziram danos em 24% e custos logísticos em 5%, atuando também na diminuição de peso de embalagens em 36% por envio [Gurumoorthy et al. 2020].

O objetivo deste trabalho é apresentar uma análise introdutória e revisional do problema da mochila 0-1, incluindo sua formulação, variantes, complexidade e métodos de resolução. A Seção 2 apresenta o referencial teórico com foco na formulação matemática, complexidade e principais abordagens de resolução.

2. Revisão Bibliográfica

O problema da mochila 0-1 pode ser formalizado como um problema de programação inteira binária. Dado um conjunto de n itens numerados de 1 a n, cada um com peso w_i e valor v_i, e dado um limite de capacidade W, busca-se uma atribuição binária $x_i \in \{0, 1\}$ (incluir ou não o item i) que maximize o valor total:

$$\begin{aligned}
\text{Objetivo:} \quad & \max \sum_{i=1}^{n} v_i x_i \\
\text{Restrição:} \quad & \sum_{i=1}^{n} w_i x_i \leq W \\
\text{Domínio:} \quad & x_i \in \{0, 1\}, \quad i = 1, \ldots, n
\end{aligned}$$

Essa formulação reflete exatamente o desafio de maximizar o valor dos itens carregados sem exceder a capacidade da mochila. Em termos práticos, escolhe-se um subconjunto de itens cujo peso total não ultrapasse W e cujo valor total seja o maior possível.

Quanto à complexidade computacional, o problema é intrinsecamente difícil. A versão de decisão ("existe subconjunto com valor pelo menos V sem exceder W?") é *NP-completa*, e a versão de otimização é *NP-difícil*. Mais especificamente, o problema da mochila 0-1 é *fraco NP-difícil* (weakly NP-hard), pois admite algoritmos pseudo-polinomiais, como o de programação dinâmica com tempo $O(nW)$ [Pisinger 2005].

Esse tempo de execução é considerado pseudo-polinomial porque depende linearmente de n e de W, mas exponencialmente do tamanho da entrada binária (já que W pode ter $\log W$ bits). Por exemplo, dobrar o valor de W dobra o tempo do algoritmo, mas o tamanho da entrada cresce apenas linearmente em $\log W$. Assim, $O(nW) = O(n2^{\log W})$ é exponencial no tamanho da entrada, o que mantém a consistência com a complexidade *NP-difícil*.

Apesar disso, diversas abordagens exatas foram desenvolvidas. A mais clássica é a própria programação dinâmica. Outras envolvem técnicas como *branch-and-bound*, onde limites obtidos por relaxação linear (como no problema da mochila fracionária) são usados para podar ramos de busca [Martello and Toth 1990]. Heurísticas que partem de soluções fracionárias e as ajustam para soluções inteiras também têm sido efetivas [Pisinger 2005].

Outro algoritmo exato relevante é o *meet-in-the-middle*, proposto por Horowitz e Sahni [Horowitz and Sahni 1974], que, embora tenha complexidade exponencial em n, pode ser mais eficiente que a programação dinâmica quando n é grande e W é moderado.

Como alternativa à ausência de algoritmos exatos de tempo polinomial, foram propostos esquemas de aproximação. Em particular, o problema admite esquemas *FPTAS* (Fully Polynomial-Time Approximation Schemes), que garantem aproximações com fator $1 + \varepsilon$ em tempo polinomial em n e $1/\varepsilon$ [Ibarra and Kim 1975, Jin 2019].

Esses algoritmos baseiam-se na discretização dos valores ou pesos, e são capazes de produzir soluções arbitrariamente próximas do ótimo em tempo viável. Além disso, heurísticas metaheurísticas como algoritmos genéticos, colônia de formigas e enxame de partículas também são aplicadas em versões de larga escala, embora sem garantias formais de aproximação.

O problema possui ainda diversas variantes que ampliam sua complexidade e aplicabilidade:

- **Mochila limitada** (bounded knapsack): permite múltiplas cópias limitadas de

cada item ($x_i \in \{0, 1, \ldots, c_i\}$);
- **Mochila ilimitada** (unbounded knapsack): número ilimitado de cópias de cada item ($x_i \in \mathbb{N}$);
- **Mochila multidimensional**: considera múltiplas restrições (como peso e volume);
- **Mochila múltipla ou de escolha múltipla**: os itens são agrupados em classes, sendo permitido escolher um item por classe;
- **Mochila quadrática e outras extensões**: envolvem funções objetivo não lineares;
- **Subset Sum**: caso especial onde $v_i = w_i$ para todos os itens, sendo exatamente *NP-completo*, como listado por Karp [Karp 1972].

Essas variantes mostram a flexibilidade do problema da mochila como modelo para cenários reais e seu papel central na teoria da complexidade e otimização combinatória.

3. Metodologia

Nesta seção, provaremos que o problema da mochila 0-1 é NP-completo e descrevemos os algoritmos utilizados para resolver o problema da mochila 0-1, com foco em três abordagens:

- Força Bruta
- Programação Dinâmica com abordagem Bottom-Up
- Programação Dinâmica com abordagem Top-Down (memoização)
- Algoritmos de Aproximação e Heurísticas

Para cada abordagem, serão apresentados:

- Funcionamento geral
- Código
- Complexidade

Implementações em C/C++ foram utilizadas para validar os resultados.

Diversas abordagens algorítmicas podem ser empregadas para resolver o Problema da Mochila 0/1, cada uma com características distintas em termos de complexidade e metodologia.

3.1. Provando que o problema da mochila 0-1 é NP-completo

Para provar que o problema da mochila 0-1 é NP-completo, podemos achar um algoritmo não determinístico em tempo polinomial e mostrar a redução para outro problema ou fazer um algoritmo verificador em tempo polinomial

Optamos por mostrar a redução do problema da mochila 0-1 para o problema de decisão do Subset Sum, que é um problema NP-completo. E mostraremos algoritmos não determinísticos que resolvem o problema da mochila 0-1 em tempo polinomial, mostrando assim que o problema da mochila 0-1 é NP-completo.

1. Intuição principal

- No SUBSET SUM, você tem uma lista de números (por exemplo, $\{2, 3, 5, 8\}$) e um alvo B. Pergunta-se: "Consigo escolher alguns números cuja soma dê exatamente B?"
- No KNAPSACK 0–1 (versão decisão), você tem itens com peso w_i e valor v_i, uma capacidade W e um mínimo de valor K. Pergunta-se: "Consigo escolher itens cujo peso total $\leq W$ e valor total $\geq K$?"
- Se fizermos $w_i = v_i = a_i$ e escolhermos $W = K = B$, as duas perguntas se tornam idênticas.

2. Montagem da instância Dada uma instância de SUBSET SUM

$$S = \{a_1, a_2, \ldots, a_n\}, \quad B \in \mathbb{Z}^+,$$

construímos a instância de KNAPSACK 0–1:

$$\big\{(w_i, v_i)\big\}_{i=1}^{n} = \big\{(a_i, a_i)\big\}_{i=1}^{n}, \quad W = B, \quad K = B.$$

Ou seja, cada número vira um item cujo peso e valor valem exatamente esse número, e a capacidade/meta da mochila é o próprio alvo.

3. Verificação com exemplo simples Considere $S = \{2, 3, 5, 8\}$ e teste $B = 3, 7, 12$:

Alvo B	**Subset Sum?**	**Knapsack?**
3	Sim, escolhe $\{3\}$	Sim, item $(3, 3)$ cabe e atinge valor 3
7	Sim, escolhe $\{2, 5\}$	Sim, itens $(2, 2) + (5, 5)$ cabem e somam valor 7
12	Não existe soma = 12	Não há combinação que alcance valor 12 sem exceder peso

Observe que, em cada caso, a resposta no SUBSET SUM coincide exatamente com a resposta em KNAPSACK 0–1. Essa construção é muito simples, executa-se em tempo polinomial e mostra que

$$\text{SUBSET SUM} \leq_p \text{KNAPSACK 0–1},$$

o que prova a NP-dificuldade do problema da mochila na sua versão decisão.

3.2. Força Bruta

A abordagem de força bruta para o problema da mochila 0-1 consiste em explorar todas as possíveis combinações de inclusão ou exclusão de cada item, avaliando recursivamente o valor total obtido em cada configuração. Para cada item, o algoritmo considera duas possibilidades: incluí-lo na mochila (caso haja capacidade suficiente) ou descartá-lo. Ao final, retorna-se o maior valor possível dentre todas as combinações viáveis. Embora conceitualmente simples, essa abordagem apresenta complexidade exponencial, $O(2^n)$, tornando-se inviável para instâncias com grande número de itens [GeeksforGeeks 2025].

Quando dizemos que a complexidade de tempo é $O(2^n)$, estamos nos referindo ao número de operações necessárias para resolver o problema no pior caso, à medida que o número de itens n aumenta.

No contexto do problema da mochila (ou problemas similares), para cada item, existem duas opções: incluí-lo ou não incluí-lo na solução. Isso significa que, para n itens, o número total de combinações possíveis é $2 \times 2 \times \ldots \times 2$ (n vezes), ou seja, 2^n.

A seguir, apresenta-se uma implementação do algoritmo de Força Bruta em C++ retirada de [GeeksforGeeks 2025]:

Listing 1. Força Bruta

```cpp
#include <bits/stdc++.h>
using namespace std;

// Returns the maximum value that
// can be put in a knapsack of capacity W
int knapsackRec(int W, vector<int> &val, vector<int> &wt, int n) {

    // Base Case
    if (n == 0 || W == 0)
        return 0;

    int pick = 0;

    // Pick nth item if it does not exceed the capacity of knapsack
    if (wt[n - 1] <= W)
        pick = val[n - 1] + knapsackRec(W - wt[n - 1], val, wt, n - 1);

    // Don't pick the nth item
    int notPick = knapsackRec(W, val, wt, n - 1);

    return max(pick, notPick);
}

int knapsack(int W, vector<int> &val, vector<int> &wt) {
    int n = val.size();
    return knapsackRec(W, val, wt, n);
}

int main() {
    vector<int> val = {1, 2, 3};
    vector<int> wt = {4, 5, 1};
    int W = 4;

    cout << knapsack(W, val, wt) << endl;
    return 0;
}
```

3.3. Programação Dinâmica - Bottom-Up

A abordagem Bottom-Up utiliza uma tabela para armazenar os resultados de subproblemas, construindo a solução de forma iterativa. O algoritmo preenche uma matriz onde cada célula representa a melhor solução possível para um determinado número de itens e capacidade da mochila. A complexidade é $O(n \cdot W)$, onde n é o número de itens e W é a capacidade da mochila.

A seguir, apresenta-se uma implementação do algoritmo Bottom-Up em C++ retirada de [GeeksforGeeks 2025]:

Listing 2. Programação Dinâmica - Bottom-Up

```cpp
#include <bits/stdc++.h>
using namespace std;

// Returns the maximum value that
// can be put in a knapsack of capacity W
int knapsack(int W, vector<int> &val, vector<int> &wt) {
    int n = wt.size();
    vector<vector<int>> dp(n + 1, vector<int>(W + 1));

    // Build table dp[][] in bottom-up manner
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= W; j++) {

            // If there is no item or the knapsack's capacity is 0
            if (i == 0 || j == 0)
                dp[i][j] = 0;
            else {
                int pick = 0;

                // Pick ith item if it does not exceed the capacity of
                   knapsack
                if(wt[i - 1] <= j)
                    pick = val[i - 1] + dp[i - 1][j - wt[i - 1]];

                // Don't pick the ith item
                int notPick = dp[i - 1][j];

                dp[i][j] = max(pick, notPick);
            }
        }
    }
    return dp[n][W];
}

int main() {
    vector<int> val = {1, 2, 3};
    vector<int> wt = {4, 5, 1};
    int W = 4;

    cout << knapsack(W, val, wt) << endl;
    return 0;
}
```

3.4. Programação Dinâmica - Top-Down (Memoização)

A abordagem Top-Down utiliza recursão com memoização para evitar o recalculo de subproblemas já resolvidos. A função recursiva verifica se a solução para um determinado estado já foi calculada e, se sim, retorna o valor armazenado. Caso contrário, calcula a solução considerando a inclusão ou exclusão do item atual. A complexidade é também $O(n \cdot W)$.

Memoização é uma técnica que armazena os resultados de subproblemas para evitar cálculos repetidos, melhorando a eficiência do algoritmo. A tabela de memoização é

preenchida conforme os subproblemas são resolvidos, permitindo que o algoritmo retorne rapidamente soluções já conhecidas.

A seguir, apresenta-se uma implementação do algoritmo Top-Down em C++ retirada de [GeeksforGeeks 2025]:

Listing 3. Programação Dinâmica - Top-Down (Memoização)

```cpp
#include <bits/stdc++.h>
using namespace std;

// Returns the maximum value that
// can be put in a knapsack of capacity W
int knapsackRec(int W, vector<int> &val, vector<int> &wt, int n,
                                        vector<vector<int>> &memo) {

    // Base Case
    if (n == 0 || W == 0)
        return 0;

    // Check if we have previously calculated the same subproblem
    if(memo[n][W] != -1)
        return memo[n][W];

    int pick = 0;

    // Pick nth item if it does not exceed the capacity of knapsack
    if (wt[n - 1] <= W)
        pick = val[n - 1] + knapsackRec(W - wt[n - 1], val, wt, n - 1,
            memo);

    // Don't pick the nth item
    int notPick = knapsackRec(W, val, wt, n - 1, memo);

    // Store the result in memo[n][W] and return it
    return memo[n][W] = max(pick, notPick);
}

int knapsack(int W, vector<int> &val, vector<int> &wt) {
    int n = val.size();

    // Memoization table to store the results
    vector<vector<int>> memo(n + 1, vector<int>(W + 1, -1));

    return knapsackRec(W, val, wt, n, memo);
}

int main() {
    vector<int> val = {1, 2, 3};
    vector<int> wt = {4, 5, 1};
    int W = 4;

    cout << knapsack(W, val, wt) << endl;
    return 0;
}
```

4. Análise de complexidade e comparação

A tabela a seguir resume a complexidade de cada abordagem:

Tabela 1. Complexidade das Abordagens

Abordagem	Complexidade de Tempo	Complexidade de Espaço
Força Bruta	$O(2^n)$	$O(n)$
Programação Dinâmica - Bottom_Up	$O(n \cdot W)$	$O(n \cdot W)$
Programação Dinâmica - Top_Down	$O(n \cdot W)$	$O(n \cdot W)$

4.1. Brute Force (Força Bruta)

Descrição

A abordagem de força bruta resolve o problema recursivamente. Para cada item, ela considera duas opções: incluí-lo na mochila (se houver capacidade) ou excluí-lo. O resultado é o máximo valor obtido entre essas duas opções.

Recorrência Algorítmica (Tempo)

$$T(n, W) = \begin{cases} 1 & \text{se } n = 0 \text{ ou } W = 0 \\ T(n-1, W) + 1 & \text{se } wt[n-1] > W \\ T(n-1, W) + T(n-1, W - wt[n-1]) + 1 & \text{se } wt[n-1] \leq W \end{cases}$$

Complexidade

Tempo: $O(2^N)$
Espaço: $O(N)$ (devido à pilha de chamadas recursivas)

4.2. Top-Down Dynamic Programming (com Memoização)

Descrição

Esta abordagem otimiza a força bruta utilizando memoização para evitar recálculos. Antes de computar um subproblema, verifica-se se o resultado já está armazenado. Se sim, retorna o valor armazenado; caso contrário, calcula e armazena.

Recorrência Algorítmica (Tempo)

Embora a estrutura seja igual à da força bruta, a memoização garante que cada subproblema (n, W) seja computado no máximo uma vez. Assim, a complexidade é:

$$T(n, W) = \begin{cases} 1 & \text{se } n = 0 \text{ ou } W = 0 \\ 1 + T(n-1, W) & \text{se } wt[n-1] > W \\ 1 + (\text{valor já memoizado ou computado uma vez}) & \text{caso geral} \end{cases}$$

Complexidade

Tempo: $O(N \cdot W)$
Espaço: $O(N \cdot W)$ para a tabela de memoização + $O(N)$ para a pilha de chamadas

4.3. Bottom-Up Dynamic Programming

Descrição

A abordagem bottom-up preenche uma tabela iterativamente, resolvendo os subproblemas menores primeiro. A solução é construída de baixo para cima, começando com 0 itens e 0 capacidade.

Recorrência Algorítmica (Tempo)

O preenchimento da tabela envolve dois laços aninhados:

$$T(n, W) = O(n \cdot W)$$

Complexidade

Tempo: $O(N \cdot W)$
Espaço: $O(N \cdot W)$. Pode ser otimizado para $O(W)$ se apenas a linha anterior for necessária.

5. Algoritmos de Aproximação e Heurísticas para o Problema da Mochila 0/1

O problema da mochila 0/1 pertence à classe dos problemas NP-completos. Essa classificação implica que, até o presente momento, não se conhece um algoritmo que seja capaz de resolvê-lo de forma exata, em tempo polinomial, para todas as possíveis instâncias. Consequentemente, à medida que o número de itens cresce, os algoritmos exatos, como a solução recursiva simples ou mesmo a programação dinâmica, se tornam computacionalmente inviáveis, especialmente em sistemas com recursos limitados ou em aplicações que exigem respostas rápidas.

Diante dessas limitações, surgem como alternativa os algoritmos de aproximação e heurísticas, que têm por objetivo encontrar soluções suficientemente boas dentro de um tempo de execução consideravelmente reduzido. Tais algoritmos não garantem a obtenção da solução ótima, mas frequentemente produzem resultados aceitáveis com significativa economia de tempo computacional.

Uma das heurísticas mais conhecidas e frequentemente utilizadas para o problema da mochila é a abordagem gulosa (greedy), baseada na razão valor/peso de cada item. A estratégia dessa heurística é selecionar prioritariamente os itens que oferecem o maior "retorno por unidade de peso", o que, intuitivamente, tende a maximizar o valor total carregado na mochila.

Algorithm 1 MochilaGulosa(capacidade, pesos, valores, n)

Para cada item i, calcule o benefício: benefício$_i = \frac{\text{valores}[i]}{\text{pesos}[i]}$
Ordene os itens em ordem decrescente de benefício
Inicialize a mochila como vazia e $valor\_total \leftarrow 0$
for cada item i na lista ordenada **do**
    if $pesos[i] \leq capacidade\_restante$ **then**
        Adicione o item i à mochila
        $valor\_total \leftarrow valor\_total + valores[i]$
        $capacidade\_restante \leftarrow capacidade\_restante - pesos[i]$
    end if
end for
return $valor\_total$

O funcionamento do algoritmo guloso pode ser descrito nas seguintes etapas:

- Cálculo da razão valor/peso: Para cada item disponível, é calculada a razão entre seu valor e seu peso, representando a eficiência relativa do item.
- Ordenação dos itens: Os itens são ordenados em ordem decrescente de razão valor/peso.
- Seleção dos itens: Os itens são inseridos na mochila, um a um, na ordem estabelecida, desde que o peso total acumulado não ultrapasse a capacidade máxima permitida (W).

Embora essa estratégia seja eficiente do ponto de vista computacional, apresentando complexidade temporal de O(n log n) devido à etapa de ordenação, ela possui limitações estruturais.

Em particular, o algoritmo guloso não é garantidamente ótimo para a versão 0/1 da mochila, pois nesta não é permitido incluir frações de um item. Assim, decisões locais baseadas na razão valor/peso podem impedir que o algoritmo alcance a combinação globalmente ótima de itens.

Por exemplo, pode haver casos em que deixar de fora um item muito eficiente localmente permite incluir outros dois menos eficientes, mas que juntos fornecem um valor total maior. Essa heurística, no entanto, se mostra bastante eficaz quando aplicada à variante racionária do problema da mochila, na qual é permitido incluir partes dos itens. Nesse caso, o algoritmo guloso fornece uma solução ótima e é amplamente adotado como método principal.

Heurísticas gulosas são ferramentas importantes em contextos onde o tempo de execução é um fator crítico e uma solução próxima da ideal é suficiente, mas é essencial compreender suas limitações para aplicá-las de forma adequada, considerando a natureza do problema e os requisitos de qualidade da solução.

6. Resultados

Nesta seção, apresentamos os resultados obtidos com as diferentes abordagens para resolver o problema da mochila 0-1.

Comparativo das Abordagens

Critério	Brute Force (Força Bruta)	Top-Down (Memoização)	Bottom-Up (Programação Dinâmica)
Estratégia	Recursiva pura	Recursiva + cache	Iterativa com tabela
Estrutura usada	Árvore binária de chamadas	Recursão + memoização	Tabela bidimensional
Recorrência	$T(n) = 2T(n-1)+1$	$T(n,W) = T(n-1,W)+T(n-1,W-wt[n-1])+1$	$T(n,W) = n \cdot W \cdot c$
Tempo	$O(2^n)$	$O(n \cdot W)$	$O(n \cdot W)$
Espaço	$O(n)$	$O(n \cdot W) + O(n)$	$O(n \cdot W)$, ou $O(W)$ com otimização
Subproblemas resolvidos	Todos (muitos repetidos)	Cada subproblema apenas uma vez	Todos iterativamente
Facilidade de implementação	Simples, porém ineficiente	Média, exige controle de cache	Mais verbosa, mas robusta
Eficiência prática	Muito baixa	Boa	Excelente
Melhor para	Problemas pequenos ou estudo conceitual	Casos moderados com repetição de subproblemas	Casos reais com grandes entradas

Resumo dos Resultados

- A abordagem de **força bruta** serve bem para fins didáticos, mas é inviável para grandes entradas devido ao crescimento exponencial de chamadas.
- A **programação dinâmica top-down com memoização** melhora drasticamente a eficiência ao evitar recomputações, mantendo uma estrutura recursiva.
- A **abordagem bottom-up** é a mais eficiente em aplicações reais, pois evita recursão e permite otimização de espaço.

Recomendação de Uso

- Use **força bruta** apenas para instâncias muito pequenas ou fins educacionais.
- Use **top-down com memoização** para problemas moderados ou quando a estrutura recursiva for preferível.
- Use **bottom-up** para problemas grandes, onde desempenho e controle de memória são essenciais.
- Use **heurístico / guloso** Quando a valocidade for prioridade e correr o risco de perda de precisão for aceitável

7. Conclusão

O problema da mochila 0-1 é um dos pilares da otimização combinatória e da teoria da complexidade computacional, servindo como referência para o estudo de algoritmos exatos, heurísticos e de aproximação. Ao longo deste trabalho, revisamos sua formulação matemática, demonstramos sua classificação como problema NP-completo e analisamos diferentes abordagens de resolução, desde métodos exatos como força bruta e programação dinâmica até heurísticas e algoritmos aproximados.

Observou-se que, embora algoritmos exatos sejam viáveis para instâncias pequenas ou moderadas, sua aplicação em cenários reais de grande escala é limitada pelo crescimento exponencial do tempo de execução. Nesses casos, heurísticas e esquemas de aproximação tornam-se alternativas práticas, fornecendo soluções suficientemente boas em tempo reduzido.

Além de sua relevância teórica, o problema da mochila 0-1 possui ampla aplicação prática em áreas como logística, finanças, corte de materiais e ciência da computação, sendo constantemente estudado e aprimorado. O contínuo desenvolvimento de novas técnicas e algoritmos reforça sua importância tanto no meio acadêmico quanto no mercado.

Portanto, compreender o problema da mochila 0-1 e suas soluções contribui para a resolução eficiente de desafios complexos em diversas áreas do conhecimento.

Referências

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

GeeksforGeeks (2025). 0-1 knapsack problem - dynamic programming. `https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/`. Acesso em: 12 jun. 2025.

Gurumoorthy, K. S., Sanyal, S., and Chaoji, V. (2020). Think out of the package: Recommending package types for e-commerce shipments. *arXiv*. Aplicado pela Amazon para reduzir em 24

Horowitz, E. and Sahni, S. (1974). Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292.

Ibarra, O. H. and Kim, C. E. (1975). Fast approximation algorithms for the knapsack and subset-sum problems. *Journal of the ACM*, 22(4):463–468.

Jin, C. (2019). An improved fptas for 0-1 knapsack. Disponível em: `https://arxiv.org/abs/1904.09562`.

Karp, R. M. (1972). Reducibility among combinatorial problems. In Miller, R. E. and Thatcher, J. W., editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York.

Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. Wiley-Interscience, Chichester.

Nemhauser, G. L. and Wolsey, L. A. (1988). *Integer and Combinatorial Optimization*. Wiley-Interscience, New York.

Pisinger, D. (2005). Where are the hard knapsack problems? *Computers & Operations Research*, 32(10):2271–2284.

Zhang, X., Qi, F., Hua, Z., and Yang, S. (2020). Solving billion-scale knapsack problems. In *Proceedings of The Web Conference 2020 (WWW)*, pages 1–8. ACM. Solver distribuído via MPI/Hadoop/Spark usado em produção diária na Ant Financial para instâncias com 1 bilhão de variáveis e restrições.