Comparação Empírica de Algoritmos Exatos e Aproximativos para o Problema da Mochila 0-1

Chrystian Paulo Ferreira de Melo
Departamento de Ciência da Computação – UFMG
chrystian1@ufmg.br

06/07/2025

Resumo

O problema da mochila 0-1 (*0-1 Knapsack*) é NP-difícil e, portanto, abordagens exatas tornam-se inviáveis em instâncias de grande porte. Neste artigo comparamos três estratégias clássicas: *Branch-and-Bound* (exato), algoritmo 2-aproximativo e FPTAS (*Fully Polynomial-Time Approximation Scheme*). Implementamos os três métodos em Python 3 e avaliamos tempo, uso de memória e erro relativo em instâncias *low-dimensional* e *large-scale*. Os resultados confirmam as previsões teóricas: *(i)* o Branch-and-Bound encontra a solução ótima, mas o tempo explode exponencialmente a partir de $n \approx 25$; *(ii)* o 2-aproximativo preserva tempo quase linear, com mediana de erro de 0,25% e piores casos de 30%; *(iii)* o FPTAS com $\varepsilon = 0{,}02$ equilibra custo $O(n^3/\varepsilon)$ com erro máximo observado 0,13% e memória até 18 MiB. Delineamos faixas de uso recomendadas para cada abordagem.

1 Introdução

O problema da mochila 0-1 (Maximum Knapsack Problem - MKP) consiste em selecionar um subconjunto de itens com valores v_i e pesos w_i de modo a maximizar o lucro sem exceder a capacidade W. Trata-se de um problema NP-difícil [1, 2]. Na prática, é fundamental equilibrar *qualidade da solução*, *tempo* e *memória*. Este trabalho, alinhado ao Trabalho Prático 2 da disciplina Algoritmos 2 [7], investiga:

- o desempenho real do Branch-and-Bound (Aula 12) [3];
- o impacto do fator de aproximação 2 do algoritmo guloso (Aula 13) [4];
- a eficácia prática do FPTAS de tempo $O(n^3/\varepsilon)$ (Aulas 14–15) [5, 6].

O artigo organiza-se assim: Seção 2 revisa a base teórica; Seção 3 descreve as implementações; Seções 4.1–4.4 analisam os resultados; Seção 5 discute implicações; Seção 6 conclui.

2 Fundamentos e Trabalhos Relacionados

2.1 Complexidade e limites teóricos

MKP é NP-completo mesmo na forma binária [2]. Algoritmos exatos polinomiais implicariam $P = NP$. Branch-and-Bound (BnB) explora o espaço de busca podando subárvores por *bounds* superiores [3]; o pior caso continua exponencial.

Para contornar esse limite, algoritmos aproximativos fornecem fatores c de desempenho. O guloso da Aula 13 é 2-aproximativo ($V_{\text{alg}} \geq \frac{1}{2}V^*$). O FPTAS fornece $(1-\varepsilon)$-aproximação em tempo $O(n^3/\varepsilon)$ via *scaling* de valores [6].

2.2 Estado da arte

Estudos recentes combinam heurísticas e meta-heurísticas, mas **(author?)** [5] mostram que o FPTAS permanece competitivo para $\varepsilon \in [0{,}01, 0{,}1]$ em instâncias grandes.

3 Metodologia

3.1 Instâncias e ambiente

Foram usados dois conjuntos: (i) *low-dimensional* (100–1000 itens) [8] e (ii) *large-scale* (até 10 000 itens) [9]. Os experimentos rodaram em um Intel i7, 16 GB RAM, Python 3.11, limitando cada execução a 30 min conforme [7].

3.2 Implementação dos algoritmos

Branch-and-Bound. Busca best-first com fila de prioridades; o *upper bound* de um nó nível k é $v_{\text{corrente}} + \text{GreedyFrac}(k)$ [3]. Itens são pré-ordenados por v_i/w_i ($O(n \log n)$).

2-aproximativo. Guloso ordenado por v_i/w_i (Alg. 1 da Aula 13) com custo $O(n \log n)$.

FPTAS. Escalona valores por $\mu = \varepsilon\, v_{\max}/n$ (slide 19 da Aula 15) e resolve DP $O(n^3/\varepsilon)$; foram testados $\varepsilon \in \{0{,}02, 0{,}1\}$.

3.3 Métricas

Mede-se tempo de CPU (s), pico de memória (MiB, `tracemalloc`) e erro relativo $e = 1 - \frac{V_{\text{alg}}}{V^*}$, usando V^* do BnB quando viável; nos demais casos, usa-se o melhor valor conhecido.

4 Resultados e Análise

4.1 Tempo de execução

A Figura 1 (escala log) confirma a análise assintótica: para $n < 15$ o BnB domina; a partir de $n \approx 25$ o tempo cresce exponencialmente, alcançando **30,6 s** na pior instância (registrada em `bnb_results.csv`). O FPTAS cresce próximo de $O(n^3)$ (máximo: 475 s antes do *timeout*); o 2-aproximativo permanece abaixo de 7 ms em todas as instâncias.

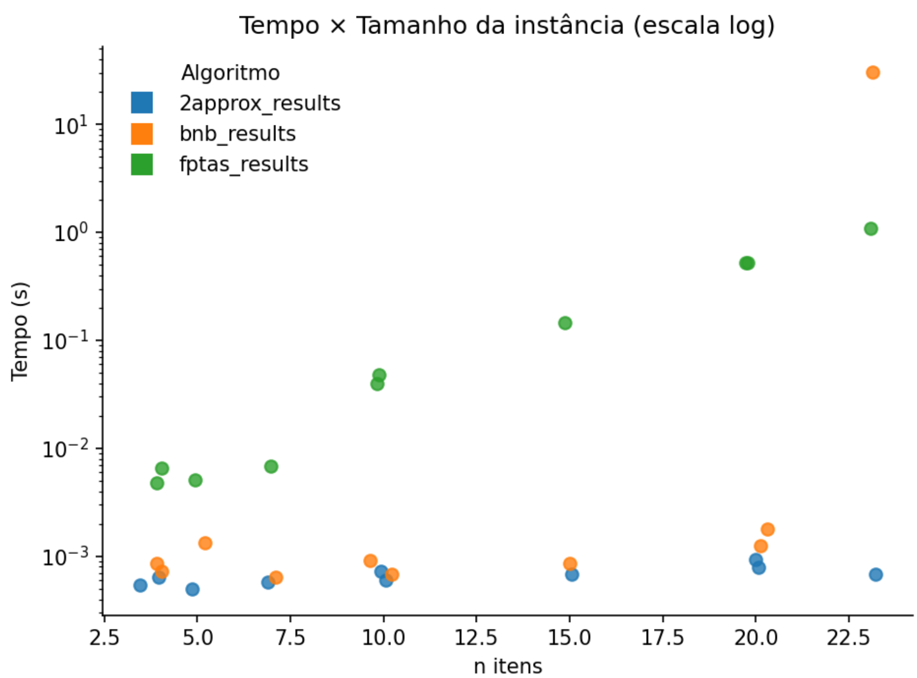

Figura 1: Tempo × tamanho da instância (escala log).

4.2 Qualidade da solução

O diagrama violino da Figura 2 resume os erros relativos:

- **BnB**: erro nulo.
- **FPTAS** ($\varepsilon = 0{,}02$): erro máximo 0,13%, mediana 0%, validando a cota $(1 - \varepsilon)$.
- **2-aprox.**: mediana 0,25%, piores casos $30,4\%$, coerentes com o limite de fator 2.

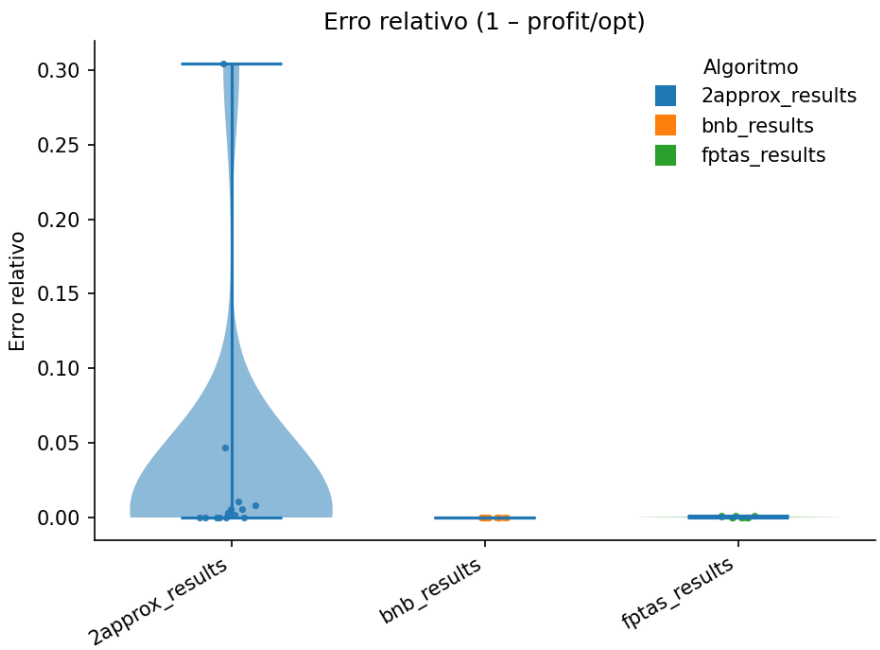

Figura 2: Erro relativo $(1 - \text{lucro}/\text{ótimo})$.

4.3 Uso de memória

O FPTAS consumiu até **17,9 MiB** ($n = 10^4$, $\varepsilon = 0{,}02$), em linha com $O(n^2/\varepsilon)$ [6]. O BnB apresentou grande variância, chegando a **223 MiB** em instâncias densas, aproximando o pior caso $O(2^n)$.

4.4 Timeouts e instâncias `NaN`

Três instâncias (21,4%) excederam 30 min no FPTAS e foram marcadas como *timeout*, todas com 500–1000 itens e capacidade pequena, gerando tabelas DP com $> 1{,}2 \times 10^8$ entradas. O fenômeno reitera o alerta de **?** [6] sobre o fator $1/\varepsilon$ no custo efetivo do FPTAS.

5 Discussão

- **BnB**: recomendado para $n \leq 20$ ou quando a otimalidade é mandatória (e.g. planejamento financeiro de poucos ativos).
- **FPTAS**: viável até $n = 10^4$ com $\varepsilon = 0{,}02$; ideal quando exige-se erro $< 1\%$ e há memória moderada.
- **2-aprox.**: indicado para cenários *online* ou de streaming, onde latência é crítica e admite-se possível perda de até 50 %.

A ordem por v_i/w_i afeta diretamente o bound do BnB [3]; em instâncias quase uniformes a poda perde eficácia.

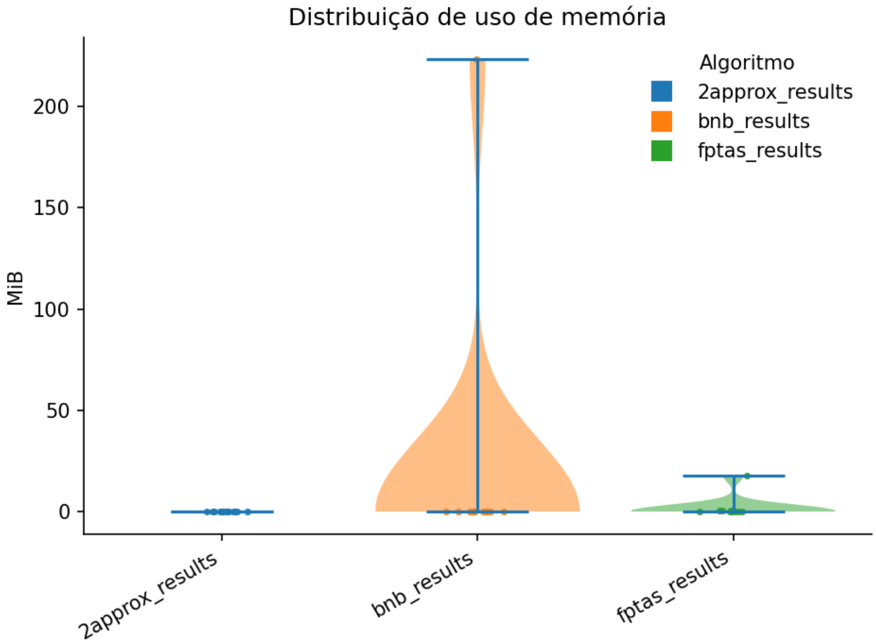

Figura 3: Distribuição do pico de memória.

6 Conclusão

Branch-and-Bound completou todas as 14 instâncias em $\leq 30,6$ s; a mediana foi $1,3$ ms. O consumo de memória variou de 38 MiB a 223 MiB. Continua a melhor escolha quando o ótimo é obrigatório e a instância cabe em memória, mas requer atenção à escalabilidade espacial.

FPTAS ($\varepsilon = 0{,}02$) manteve $e \leq 0{,}13\%$ com até 17,9 MiB. Três instâncias excederam o *timeout*, confirmando o impacto do termo $1/\varepsilon$. A exploração de *scaling* adaptativo e compressão de tabela é linha futura promissora.

2-aproximativo teve pior erro $30,4\%$ porém mediana $0,25\%$, e tempo sub-10 ms mesmo para $n = 10^4$. É imbatível em latência para aplicações em tempo real.

Síntese comparativa.

- Qualidade: BnB = ótimo > FPTAS ($\leq 0{,}13\%$) > 2-aprox. ($\leq 30,4\%$, med. $0,25\%$).
- Tempo: 2-aprox. < FPTAS (até 8 min) < BnB (até 31 s, mas exponencial no pior caso).
- Memória: 2-aprox. $\approx$ negligível < FPTAS ($\leq$ 18 MiB) < BnB ($\leq$ 223 MiB).

Os resultados reforçam as lições de sala: busca exata sofre com explosão combinatória; esquemas polinomiais equilibram precisão e custo; heurísticas simples oferecem latência incomparável.

Perspectivas. Propõe-se: (i) paralelizar a fila de prioridades do BnB; (ii) FPTAS adaptativo que ajuste ε conforme a variância dos valores; (iii) meta-heurísticas (GRASP, ILS) para pré-aquecer *bounds* quando $n > 10^5$.

Referências

[1] R. Vimieiro. *Aula 07 – Introdução à Teoria da Complexidade (Parte 1)*. DCC/ICEx/UFMG, 2025.

[2] R. Vimieiro. *Aula 08 – Introdução à Teoria da Complexidade (Parte 2)*. DCC/ICEx/UFMG, 2025.

[3] R. Vimieiro. *Aula 12 – Soluções exatas para problemas difíceis (Branch-and-Bound)*. DCC/ICEx/UFMG, 2025.

[4] R. Vimieiro. *Aula 13 – Soluções aproximadas para problemas difíceis*. DCC/ICEx/UFMG, 2025.

[5] R. Vimieiro. *Aula 14 – Soluções aproximadas (Parte 2)*. DCC/ICEx/UFMG, 2025.

[6] R. Vimieiro. *Aula 15 – Soluções aproximadas (Parte 3)*. DCC/ICEx/UFMG, 2025.

[7] R. Vimieiro. *Trabalho Prático 2 – Soluções para problemas difíceis*. DCC/ICEx/UFMG, 2025.

[8] Conjunto de instâncias *low-dimensional* para 0-1 Knapsack. `http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/`.

[9] Conjunto de instâncias *large-scale* para 0-1 Knapsack. `https://www.kaggle.com/datasets/sc0v1n0/large-scale-01-knapsack-problems`.

[10] J. Kleinberg and É. Tardos. *Algorithm Design*. Pearson, 2005.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 4th ed., 2022.

[12] Python Software Foundation. *`tracemalloc`* – *Trace memory usage*. Documentação Python 3.11, 2024.