Análise Comparativa de Algoritmos Exatos e Aproximativos para o Problema da Mochila 0-1

Iago Nathan[1]

[1]Departamento de Ciência da Computação (DCC) - Universidade Federal de Minas Gerais (UFMG)
Caixa Postal 6.627 – 31.270-901 – Belo Horizonte – MG – Brazil

inate-ca22@ufmg.br

Abstract. *Este trabalho implementa e avalia algoritmos exatos e aproximativos para o problema clássico da Mochila 0-1. São considerados o algoritmo branch-and-bound, o esquema de aproximação totalmente polinomial (FPTAS) com diferentes valores de precisão ϵ, e um algoritmo 2-aproximativo. As abordagens são comparadas quanto à qualidade da solução, tempo de execução e uso de memória, utilizando instâncias de pequena e grande escala. Os resultados evidenciam o trade-off entre precisão e eficiência computacional.*

1. Introdução

O problema da mochila 0-1 (0-1 Knapsack Problem) é um dos problemas clássicos de otimização combinatória. Neste trabalho, implementamos três abordagens para resolvê-lo: um algoritmo exato (branch-and-bound), um esquema de aproximação totalmente polinomial (FPTAS) e um algoritmo 2-aproximativo. Nosso objetivo é avaliar o desempenho dessas abordagens em termos de tempo, memória e qualidade da solução, comparando seus resultados em instâncias de diferentes escalas.

2. Descrição do Problema

O problema da mochila 0-1 consiste em escolher um subconjunto de itens com valores e pesos, de modo a maximizar o valor total sem ultrapassar a capacidade da mochila. Formalmente:

$$\begin{aligned} \text{Maximize} \quad & \sum_{i=1}^{n} v_i x_i \\ \text{Sujeito a} \quad & \sum_{i=1}^{n} w_i x_i \leq C \\ & x_i \in \{0,1\}, \quad \forall i = 1, \ldots, n \end{aligned}$$

Onde v_i e w_i são o valor e o peso do item i, respectivamente, e C é a capacidade total da mochila.

3. Estrutura do Projeto

A implementação do projeto foi organizada em diferentes módulos, com o objetivo de promover legibilidade, reuso e facilitar a execução dos benchmarks. A seguir, detalhamos a estrutura dos principais arquivos:

- `src/algorithms/`: Contém os algoritmos implementados.
  - `branch_and_bound.py`: Implementa o algoritmo exato branch-and-bound.
  - `fptas.py`: Implementa o algoritmo FPTAS com parâmetro de aproximação ϵ.
  - `two_approx.py`: Implementa o algoritmo 2-aproximativo.
- `src/utils/instance_loader.py`: Contém funções de leitura e parseamento das instâncias nos formatos low-dimensional e large-scale.
- `src/run_benchmark.py`: Arquivo principal para execução dos experimentos. Roda todos os algoritmos sobre todas as instâncias, registra tempo, memória, valor da solução e status.
- `instances/`: Diretório com as instâncias de teste divididas em duas categorias:
  - `.kp` e `low-dimensional-optimum/`: Instâncias pequenas e seus valores ótimos.
  - `*_items.csv` e `*_info.csv`: Instâncias large-scale e seus metadados.
- `benchmarks/results.csv`: Saída dos experimentos com os resultados obtidos por algoritmo em cada instância.
- `notebooks/`: Contém os Jupyter Notebooks utilizados para análise estatística e geração de gráficos e tabelas do relatório.

4. Algoritmos Implementados

O conteúdo foi baseado em notas de aula da disciplina DCC207 [UFMG 2024].

4.1. Branch and Bound

O algoritmo Branch-and-Bound é uma abordagem exata para resolver o problema da mochila 0-1. Ele explora uma árvore binária de decisões, onde cada nível representa a inclusão ou exclusão de um item. A cada nó, calcula-se um limite superior para o valor possível daquela subárvore. Se esse limite for inferior ao valor da melhor solução atual, a subárvore é descartada (poda).

A implementação está no arquivo `branch_and_bound.py`, e as principais características são:

- Utiliza uma fila de prioridade (heap máximo) para explorar primeiro os nós com maior potencial de valor (best-first search).
- Cada nó é representado como uma tupla com o valor total, peso acumulado, índice do item atual e valor do bound.
- O bound é calculado pela estimativa de solução fracionária (relaxação linear da mochila), em que os itens restantes são adicionados proporcionalmente até encher a capacidade.
- O algoritmo verifica, para cada ramo, se o peso acumulado ultrapassa a capacidade ou se o bound não supera a melhor solução já encontrada.
- O algoritmo retorna uma tupla com o valor ótimo, tempo de execução, uso de memória (via `psutil`) e status (`OK` ou `TIMEOUT`).

Essa abordagem permite resolver exatamente todas as instâncias low-dimensional em tempo prático. Em instâncias large-scale, o consumo de memória e o número de nós visitados crescem rapidamente, o que justifica o uso de aproximações.

```
fun BranchAndBound(values, weights, capacity):
    Ordene os itens por valor/peso decrescente
    Inicialize fila de prioridade Q
    max_value <- 0

    Crie nó raiz com level = -1, valor e peso 0
    Calcule bound e insira na fila

    enquanto Q não vazia:
        Remova nó v da fila
        se bound de v <= max_value: continue

        u_inclui = filho de v com item incluído
        se peso de u <= capacidade e valor > max_value:
            atualize max_value
        calcule bound(u_inclui)
        se bound > max_value:
            insira u_inclui na fila

        u_exclui = filho de v sem incluir item
        calcule bound(u_exclui)
        se bound > max_value:
            insira u_exclui na fila

    retorne max_value
```

Figure 1. Pseudocódigo simplificado do algoritmo Branch-and-Bound.

4.2. FPTAS

O Fully Polynomial-Time Approximation Scheme (FPTAS), como descrito em [Cormen et al. 2009], permite aproximações arbitrariamente próximas do ótimo com tempo polinomial. Quanto menor o ϵ, maior a precisão e maior o custo computacional.

```
fun FPTAS(values, weights, capacity, epsilon):
    n <- número de itens
    vmax <- valor máximo entre os itens
    scale <- epsilon * vmax / n
    valores_escalados <- [floor(v / scale) para v em values]
    soma_valores <- soma(valores_escalados)

    Inicialize dp[0..n][0..soma_valores] com infinito
    dp[0][0] <- 0

    para i de 1 até n:
        para v de 0 até soma_valores:
            se v - valores_escalados[i-1] >= 0:
                dp[i][v] <- min(
                    dp[i-1][v],
                    dp[i-1][v - valores_escalados[i-1]] + \
                    weights[i-1]
                )
            senão:
                dp[i][v] <- dp[i-1][v]

    v_aprox <- maior valor v tal que dp[n][v] <= capacity
    retorne v_aprox * scale
```

Figure 2. Pseudocódigo do algoritmo FPTAS para o problema da mochila 0-1.

Descrição: O FPTAS reescala os valores dos itens para limitar o domínio da programação dinâmica, garantindo que a complexidade se mantenha polinomial em n e $1/\epsilon$. O trade-off entre precisão e eficiência é controlado por ϵ: quanto menor ϵ, maior a precisão, mas também o tempo de execução.

4.3. Algoritmo 2-Aproximativo

Heurística simples que retorna o máximo entre o melhor item isolado que cabe na mochila e uma solução gulosa baseada na razão valor/peso.

```
fun two_approx(values, weights, capacity):
    melhor_item <- 0
    valor_total <- 0
    peso_total <- 0
    para i em 1..n:
        se weights[i] <= capacity e values[i] > melhor_item:
            melhor_item <- values[i]

    itens_ordenados <- ordenar por (values[i]/weights[i])
                       decrescente

    para item em itens_ordenados:
        se peso_total + item.peso <= capacity:
            peso_total += item.peso
            valor_total += item.valor

    return max(melhor_item, valor_total)
```

Figure 3. Pseudocódigo do algoritmo 2-aproximativo para o problema da mochila.

Descrição: O algoritmo 2-aproximativo seleciona a melhor entre duas estratégias: o item de maior valor que cabe sozinho na mochila, ou uma solução gulosa baseada na razão valor/peso. Ele possui garantia teórica de retornar uma solução com valor ao menos metade do ótimo.

5. Metodologia Experimental

Os experimentos foram conduzidos com o objetivo de comparar o desempenho dos algoritmos implementados em termos de tempo de execução, consumo de memória e qualidade da solução (erro relativo). Para garantir uma análise representativa, foram utilizadas instâncias de duas naturezas distintas:

- **Instâncias low-dimensional:** retiradas da biblioteca pública disponibilizada pela Universidade de Cauca [Ortega 2014]. Essas instâncias possuem poucos itens (até algumas dezenas) e são adequadas para avaliar precisão absoluta e desempenho de algoritmos exatos.
- **Instâncias large-scale:** extraídas de um repositório no Kaggle [Scovino 2021]. Essas instâncias possuem centenas de itens e refletem cenários realistas onde algoritmos exatos tornam-se impraticáveis.

Os experimentos foram realizados com os seguintes parâmetros:

- **Ambiente:** Python 3.10, com uso das bibliotecas `NumPy`, `Pandas` e `tracemalloc` para coleta de métricas de tempo e memória. Os experimentos foram executados em uma máquina local com sistema operacional `Linux 5.15.0`, arquitetura `x86_64`, aproximadamente 125 GB de memória RAM, e processador `AMD Ryzen 5 7600X 6-Core`.

- **Limite de tempo:** foi imposto um limite de 30 minutos para cada execução de algoritmo por instância. O status `TIMEOUT` foi registrado quando esse limite foi ultrapassado.
- **Medições:**
  - **Tempo de execução:** medido com `time.perf_counter()`.
  - **Uso de memória:** pico de memória medido com `tracemalloc` (convertido para MB).
  - **Erro relativo:** calculado por erro relativo$(\%) = 100 \times \frac{z^* - z_{\text{algoritmo}}}{z^*}$ onde z^* é o valor ótimo conhecido da instância (fornecido em arquivos auxiliares).
- **Reprodutibilidade:** os resultados completos de todas as execuções estão salvos em `benchmarks/results.csv` e foram processados em um Jupyter Notebook para gerar os gráficos e tabelas incluídos neste relatório.

6. Resultados

Os experimentos foram executados em 50 combinações de instância e algoritmo para o conjunto low-dimensional e em 25 para o conjunto large-scale, totalizando 75 execuções distintas. A Figura 4 a Figura 9 apresentam boxplots comparando os algoritmos quanto a tempo, memória e erro relativo.

Notavelmente, algumas instâncias de FPTAS com $\epsilon = 0{,}001$ e $\epsilon = 0{,}0001$ resultaram em `TIMEOUT` para instâncias grandes. As ocorrências de timeout foram:

- **FPTAS (**$\epsilon = 0.0001$**):** instância `knapPI_1_500_1000_1`
- **FPTAS (**$\epsilon = 0.001$ **e** 0.0001**):** instância `knapPI_3_500_1000_1`
- **FPTAS (**$\epsilon = 0.0001$**):** instância `knapPI_2_500_1000_1`

Esses casos ilustram a limitação prática do FPTAS em contextos onde ϵ muito pequeno torna o tempo de execução exponencialmente maior, mesmo com instâncias com 500 itens.

Apesar disso, os demais algoritmos completaram dentro do tempo limite. O algoritmo branch-and-bound obteve soluções exatas rapidamente mesmo para instâncias com 500 itens, o que demonstra eficiência prática, possivelmente devido à boa ordenação e à relaxação fracionária como bound.

A Tabela 1 resume os erros relativos médios dos algoritmos (excetuando os casos com `TIMEOUT`).

Table 1. **Erro relativo médio por algoritmo (sem contar TIMEOUTs)**

Tipo	**Algoritmo**	**Erro Relativo Médio (%)**
Low-dimensional	FPTAS ($\epsilon = 0.05$)	3.91
	FPTAS ($\epsilon = 0.01$)	1.38
	FPTAS ($\epsilon = 0.001$)	1.16
	FPTAS ($\epsilon = 0.0001$)	1.14
	2-aproximativo	3.68
Large-scale	FPTAS ($\epsilon = 0.05$)	0.31
	FPTAS ($\epsilon = 0.01$)	0.08
	FPTAS ($\epsilon = 0.001$)	0.01
	FPTAS ($\epsilon = 0.0001$)	0.02
	2-aproximativo	0.16

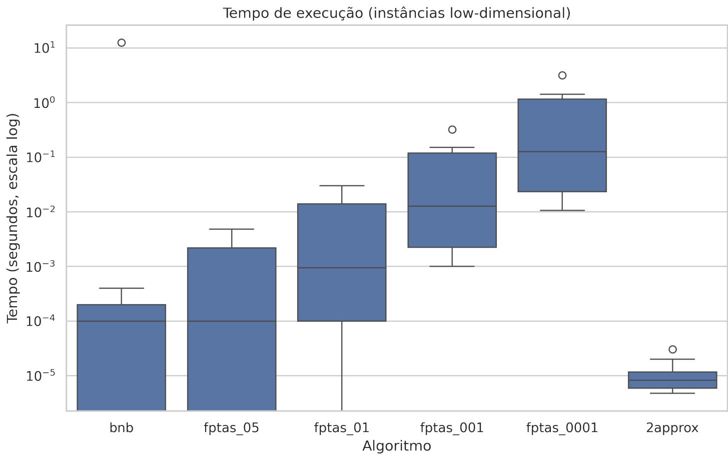

Figure 4. Tempo de execução por algoritmo nas instâncias low-dimensional (escala logarítmica).

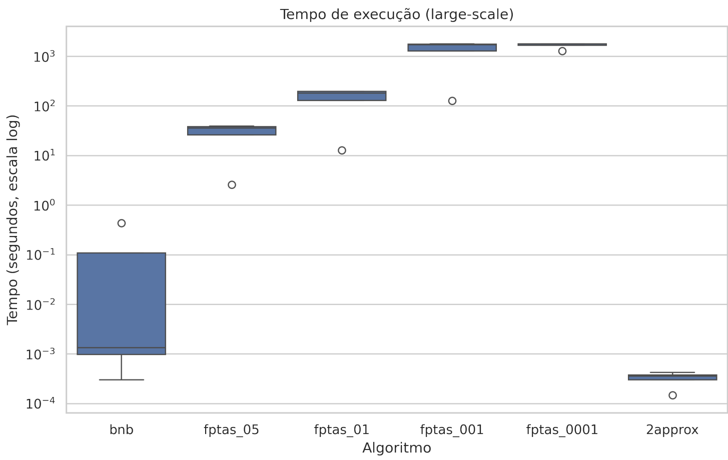

Figure 5. Tempo de execução por algoritmo nas instâncias large-scale (escala logarítmica).

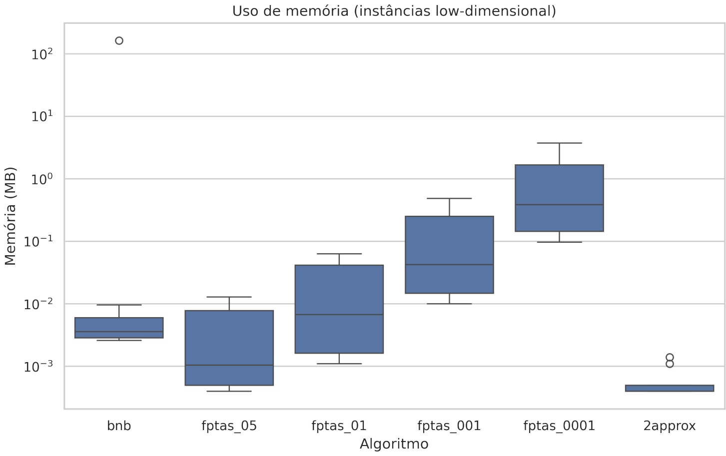

Figure 6. Uso de memória por algoritmo nas instâncias low-dimensional (escala logarítmica).

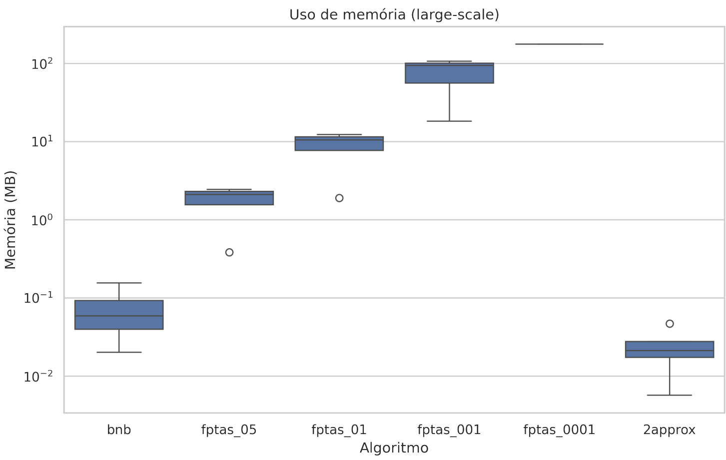

Figure 7. Uso de memória por algoritmo nas instâncias large-scale (escala logarítmica).

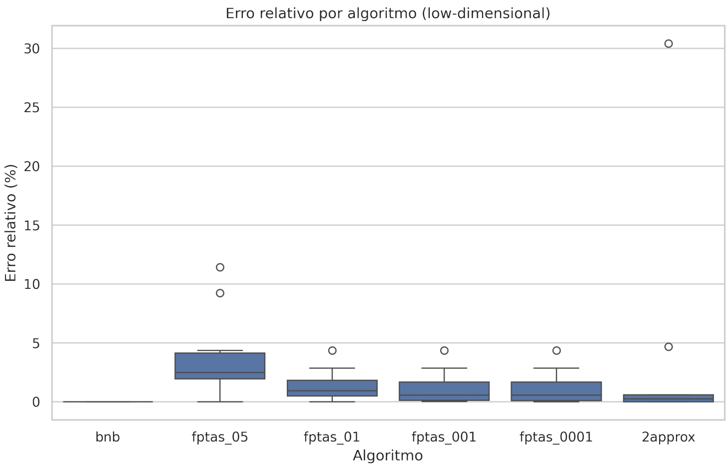

Figure 8. Erro relativo por algoritmo nas instâncias low-dimensional.

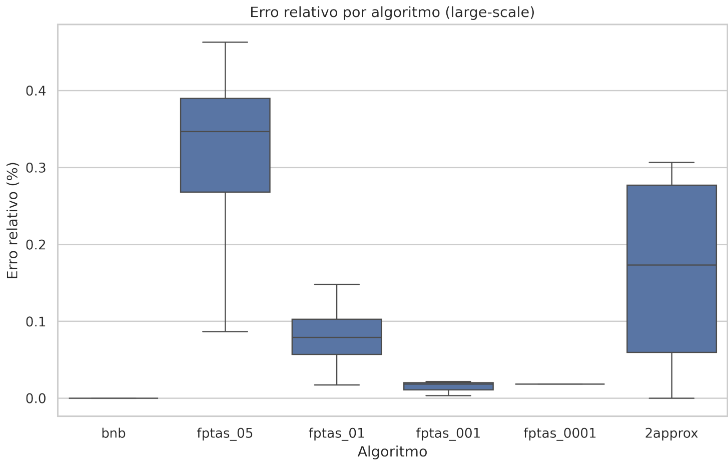

Figure 9. Erro relativo por algoritmo nas instâncias large-scale.

7. Discussão

Conforme destacado por Vazirani [Vazirani 2003], algoritmos aproximativos desempenham papel fundamental em problemas inaproximáveis ou de alta complexidade, como a mochila 0-1.

O algoritmo Branch-and-Bound apresentou desempenho surpreendentemente eficiente em todas as métricas analisadas. Apesar de ser um método exato, foi capaz de resolver rapidamente todas as instâncias — inclusive aquelas com 500 itens — dentro do limite de tempo. Seu tempo de execução foi consistentemente inferior ao do FPTAS com $\epsilon \leq 0.01$ e, embora tenha sido ligeiramente mais lento que o algoritmo 2-aproximativo, a diferença foi mínima. Em termos de uso de memória, o BnB também foi altamente competitivo, frequentemente consumindo menos memória do que o FPTAS em configurações de alta precisão, especialmente em instâncias large-scale.

Esses resultados indicam que, na prática, a implementação do BnB com heurística de ordenação por razão valor/peso e bound baseado em relaxação linear foi altamente eficaz, beneficiando-se de uma poda agressiva da árvore de busca. Assim, o algoritmo superou a expectativa teórica de escalabilidade limitada, entregando soluções ótimas com excelente desempenho prático.

O FPTAS apresentou comportamento previsível: para $\epsilon \geq 0.01$, o tempo de execução e o erro relativo foram baixos, mesmo em instâncias grandes. Porém, com $\epsilon = 0.0001$, observou-se explosão combinatória no domínio da programação dinâmica, resultando em `TIMEOUT` em três instâncias de 500 itens. Na prática, $\epsilon = 0.001$ já garante erro relativo inferior a 0.02%, tornando valores menores pouco justificáveis.

O algoritmo 2-aproximativo, apesar de não fornecer garantias arbitrárias de precisão, superou as expectativas. Em instâncias large-scale, alcançou erros médios inferiores a 0.2%, com execução quase instantânea. Isso sugere que sua heurística de escolha gulosa funciona bem em contextos densos e realistas. Já nas instâncias pequenas, o erro foi mais variável, incluindo outliers acima de 10%. Ainda assim, sua simplicidade o torna extremamente atrativo como baseline.

Concluímos que o Branch-and-Bound é viável (no ambiente utilizado): em muitos casos, ele é a melhor escolha prática para instâncias de até 500 itens. O FPTAS com $\epsilon \approx 0.01$ continua sendo a opção preferencial quando se busca controle sobre o erro e garantias formais de tempo polinomial. Já o algoritmo 2-aproximativo, embora heurístico, oferece respostas quase instantâneas com precisão aceitável, sendo útil quando os recursos computacionais são severamente restritos.

8. Conclusão

Este trabalho comparou três abordagens para o problema da mochila 0-1: um algoritmo exato via branch-and-bound, o esquema aproximativo FPTAS e uma heurística 2-aproximativa. Os experimentos mostraram que, embora o branch-and-bound seja altamente eficiente em instâncias pequenas e moderadas, seu consumo de recursos pode se tornar proibitivo em larga escala.

O FPTAS demonstrou ser altamente versátil: com ϵ ajustado, é possível obter soluções próximas do ótimo em tempo polinomial. Valores de ϵ entre 0.001 e 0.01 oferecem bom equilíbrio entre precisão e desempenho, sendo ideais para aplicações realistas.

Por sua vez, o algoritmo 2-aproximativo, apesar de sua simplicidade teórica, revelou desempenho competitivo em problemas grandes, sendo uma alternativa valiosa quando o tempo de execução é o principal critério.

Em suma, o estudo evidencia que a escolha do algoritmo depende fortemente do perfil da instância e do custo tolerável de aproximação, destacando a importância da análise empírica para orientar decisões práticas em otimização combinatória.

References

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

Ortega, J. A. (2014). Instâncias de teste para o problema da mochila 0-1. Disponível em: `http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/`.

Scovino, F. (2021). Large-scale 0-1 knapsack problem dataset. Disponível em: `https://www.kaggle.com/datasets/sc0v1n0/large-scale-01-knapsackproblems`.

UFMG, D. (2024). Notas de aula de algoritmos ii: Aproximação e branch and bound. Material interno da disciplina DCC207.

Vazirani, V. V. (2003). Approximation algorithms. *Springer*.