Trabalho Prático 2 – Algoritmos 2
Comparação de Abordagens Algorítmicas para o 0-1 Knapsack Problem

Lucas Paulo de Oliveira Silva - 2022043469
Wesley Marques Daniel Chaves - 2020053246

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brazil

lucaspaulo1@hotmail.com, wesleymarques@dcc.ufmg.br

Abstract. *This practical work addresses the classical 0-1 knapsack problem, focusing on the implementation and comparison of three solution strategies: an exact branch-and-bound algorithm, a Fully Polynomial-Time Approximation Scheme (FPTAS), and a custom 2-approximation algorithm. Implementations were performed with analysis of the trade-offs between solution quality and computational performance. Experimental results provide insights into the suitability of each approach for different problem instances, highlighting their theoretical guarantees and practical applicability.*

Resumo. *Este trabalho prático aborda o clássico problema da mochila 0-1, com foco na implementação e comparação de três estratégias de solução: um algoritmo exato branch-and-bound, um esquema aproximativo FPTAS (Fully Polynomial-Time Approximation Scheme) e um algoritmo 2-aproximativo desenvolvido por nós. As implementações foram realizadas com análise do equilíbrio entre qualidade da solução e desempenho computacional. Resultados experimentais fornecem insights sobre a adequação de cada abordagem para diferentes instâncias do problema, destacando garantias teóricas e aplicabilidade prática.*

1. Introdução

O problema da mochila binária, também conhecido como 0-1 knapsack, consiste em selecionar um subconjunto de itens, cada um com peso e valor, de modo a maximizar o valor total sem exceder uma dada capacidade. Essa formulação clássica é NP-difícil, o que implica que algoritmos exatos tendem a ter custo exponencial no pior caso, restringindo sua aplicação a instâncias de tamanho moderado. Ainda assim, em muitas situações práticas, é desejável obter soluções com garantia de qualidade ou que se aproximem do ótimo em tempo aceitável. Com base nessas considerações, foram implementadas e comparadas três estratégias distintas: um algoritmo exato via branch-and-bound, um esquema aproximativo baseado em FPTAS (Fully Polynomial-Time Approximation Scheme) e um algoritmo 2-aproximativo formulado e desenvolvido no escopo deste estudo.

Este relatório técnico apresenta as decisões de projeto e implementação, os ambientes de teste utilizados, a análise crítica dos resultados obtidos e uma discussão sobre os contextos mais adequados para o uso de cada abordagem. A seguir, são detalhados os componentes do estudo, com foco no desenvolvimento das soluções, suas particularidades e os principais pontos observados durante os experimentos.

2. Conceitos básicos

O problema da mochila 0-1 pode ser formalizado da seguinte forma: dado um conjunto de n itens, cada item i caracterizado por um peso $w_i \in N$ e um valor $v_i \in N$, e uma capacidade máxima $W \in N$, determine um vetor binário $x = (x_1, \ldots, x_n)$, com $x_i \in \{0, 1\}$, que maximize a função objetivo:

$$\max \sum_{i=1}^{n} v_i x_i$$

sujeito à restrição:

$$\sum_{i=1}^{n} w_i x_i \leq W$$

Essa modelagem expressa a escolha exata de cada item: $x_i = 1$ indica que o item i foi incluído na mochila, enquanto $x_i = 0$ indica que ele foi descartado. A restrição de capacidade impõe que a soma dos pesos não ultrapasse W, refletindo o limite físico ou orçamentário do problema.

Em termos de complexidade, a mochila 0-1 está entre os primeiros problemas classificados como NP-difíceis. Isso significa que não se conhece algoritmo exato que seja polinomial no pior caso em n, e que instâncias de tamanho moderado podem exigir tempo exponencial para serem resolvidas com garantia de ótimo. Por outro lado, existem técnicas de programação dinâmica que resolvem o problema em tempo pseudo-polinomial:

$$O(nW)$$

eficientes quando W não cresce demasiadamente, e esquemas de aproximação que reduzem drasticamente o custo computacional em troca de garantias sobre a qualidade da solução.

Entre as técnicas clássicas para atacar a mochila 0-1, destacam-se:

1. **Programação dinâmica exata**, que preenche gradativamente uma matriz de dimensões $(n + 1) \times (W + 1)$ para obter o valor ótimo.

2. **Heurísticas gulosas**, que ordenam itens pelo quociente valor/peso (densidade) e inserem-nos até a capacidade, fornecendo soluções rápidas, porém sem garantia de proximidade com o ótimo.

3. **Branch-and-bound**, que explora uma árvore de decisões e utiliza estimativas de bound — em particular o bound fracionário, que supõe possibilidade de fracionar o último item para preencher exatamente a capacidade — a fim de podar ramos sem potencial de gerar soluções melhores.

Além disso, esquemas de aproximação como o **FPTAS** garantem, para dado ε, uma solução com valor ao menos: $(1 - \varepsilon)$ vezes o ótimo, em tempo polinomial tanto em n quanto em $1/\varepsilon$, tornando-se poderosos em cenários onde pequenas perdas de qualidade são aceitáveis.

Com esses conceitos bem estabelecidos, podemos avançar para descrever como cada uma das abordagens — branch-and-bound exato, FPTAS aproximativo e algoritmo 2-aproximativo — foi projetada e implementada, discutindo as escolhas de estrutura de dados, a lógica de bound e as decisões de busca que permitiram obter tanto eficiência quanto rigor teórico.

3. Descrição das Abordagens

Inicialmente, todas as soluções foram desenvolvidas na linguagem C++ , em versões que permitiram validação e coleta de métricas, como tempo e uso de memória. As implementações finais foram ajustadas para otimização e avaliação de desempenho, mantendo a semântica dos algoritmos.

Nesta seção, descrevemos cada abordagem exata e aproximativa, detalhando a estimativa de custo adotada, as estruturas de dados escolhidas e as motivações por trás das decisões de projeto.

3.1. Branch-and-Bound

A implementação do branch-and-bound segue estritamente os conceitos apresentados nos materiais da disciplina. Inicialmente, cada item é representado por seus valores de peso e valor, além da razão v_i/w_i, e os itens são ordenados em ordem decrescente dessa razão.

Para um nó u da árvore de busca, correspondente a um estado com nível i, valor acumulado $u.value$ e peso acumulado $u.weight$, calcula-se uma estimativa superior do valor ótimo possível a partir desse nó, chamada *bound*, definida por

$$\text{bound}(u) = u.value + (C - u.weight) \times \frac{v_{i+1}}{w_{i+1}},$$

onde C representa a capacidade da mochila e (v_{i+1}, w_{i+1}) são o valor e peso do próximo item a ser considerado.

Essa estimativa usa o *bound fracionário*, que considera a possibilidade de fracionar o próximo item para preencher exatamente a capacidade restante, garantindo um cálculo simples em tempo $O(1)$ por nó e permitindo uma poda eficaz dos ramos da árvore que não podem levar a soluções melhores.

A estratégia de exploração adotada é a busca *best-first*, apoiada por uma fila de prioridade (`std::priority_queue`) que mantém os nós ordenados de acordo com seus valores de bound. Esse tipo de estrutura permite selecionar em tempo logarítmico o nó mais promissor a cada passo, garantindo que o caminho explorado seja o que possui maior potencial de levar à solução ótima. Diferentemente da busca em profundidade (*depth-first*), essa abordagem permite realizar podas precoces e evitar a exploração de subárvores que não possam melhorar a melhor solução encontrada até o momento.

Cada nó é representado por uma estrutura contendo o nível na árvore, o valor acumulado, o peso acumulado e o bound correspondente. A criação dos nós filhos ocorre por dois ramos de decisão: inclusão ou exclusão do próximo item. Quando a inclusão é viável (i.e., o peso resultante não excede a capacidade), o nó filho é criado e considerado para expansão futura somente se seu bound superar o valor ótimo atual. O mesmo critério de bound é utilizado para decidir pela exclusão do item.

O nó raiz é definido com nível -1, valor e peso igual a zero, e bound inicial calculado com base nos itens ordenados. A partir desse ponto, o algoritmo realiza expansões sucessivas até que todos os nós promissores sejam avaliados ou podados. A condição de parada ocorre quando a fila de prioridade está vazia ou quando o bound de um nó não ultrapassa o valor ótimo atual.

Essa implementação busca o equilíbrio entre precisão e desempenho. Embora garanta a obtenção da solução ótima, ela pode demandar tempo e memória elevados para instâncias de grande porte, dada a natureza combinatória do problema. No entanto, o uso estratégico da estimativa fracionária e da busca best-first contribui para reduzir substancialmente o espaço de busca explorado.

3.2. Algoritmo Aproximativo

O algoritmo aproximativo implementado utiliza o esquema Fully Polynomial-Time Approximation Scheme (FPTAS), que garante uma solução com valor mínimo igual a $(1-\varepsilon)$ vezes o ótimo, para um dado parâmetro $\varepsilon > 0$. A técnica principal consiste no arredondamento dos valores dos itens por um fator de escala, reduzindo o domínio da programação dinâmica e permitindo resolução em tempo polinomial em função de n e $1/\varepsilon$.

Dado o maior valor dos itens $v_{\max} = \max_i v_i$, calcula-se o fator de escala

$$\mu = \frac{\varepsilon \cdot v_{\max}}{n}$$

e os valores são arredondados para $v'_i = \left\lfloor \frac{v_i}{\mu} \right\rfloor$.

A programação dinâmica é usada para preencher uma tabela onde $dp[i][v]$ representa o peso mínimo necessário para alcançar o valor arredondado v utilizando os primeiros i itens, respeitando a capacidade máxima W.

Ao final, a solução aproximada é dada por

$$\text{approx\_value} = \mu \times \max\{v : dp[n][v] \leq W\},$$

com garantia de $\text{approx\_value} \geq (1-\varepsilon) \cdot \text{opt}$.

O parâmetro ε possibilita controlar o equilíbrio entre qualidade da solução e custo computacional, tornando o FPTAS adequado para cenários que exigem soluções próximas do ótimo com eficiência prática.

A implementação em C++ apresentada consiste em um algoritmo que realiza uma normalização dos valores dos itens utilizando um fator de escala $\mu = \frac{\varepsilon \cdot v_{\max}}{n}$, onde $v_{\max}$ é o maior valor entre os itens. Os valores escalonados são arredondados para baixo, reduzindo a faixa de valores possíveis e, assim, o tamanho da tabela de programação dinâmica. Em seguida, uma matriz dp é inicializada para armazenar o peso mínimo necessário para obter cada soma de valores escalonados, respeitando a capacidade máxima da mochila. O algoritmo percorre todos os itens e, para cada um, decide entre incluí-lo ou não, atualizando a tabela com o peso acumulado mínimo. Por fim, determina-se o maior valor arredondado viável e aplica-se a fator de escala para obter o valor aproximado final.

A complexidade de tempo desta implementação é $O(n^3/\varepsilon)$, uma vez que o somatório dos valores escalonados é limitado por n^2/ε. Já a complexidade de espaço

também é $O(n^2/\varepsilon)$, pois a tabela dp possui dimensão $(n+1) \times (v'_{sum}+1)$. Dessa forma, o algoritmo oferece uma aproximação arbitrariamente próxima do ótimo com tempo polinomial em n e $1/\varepsilon$, caracterizando-o como um FPTAS.

3.3. Algoritmo 2-Aproximativo

Propomos uma heurística de fator-2 para o problema da Mochila 0-1 que combina duas estratégias simples e retorna o maior valor entre elas. A primeira estratégia escolhe o único item de maior valor que cabe sozinho na mochila, denotado por V_1. A segunda realiza uma construção gulosa por densidade: para cada item calcula-se $d_i = v_i/w_i$, ordena-se em ordem decrescente de d_i e insere-se cada item enquanto couber na capacidade residual, acumulando valor total V_2. A solução final é $V = \max\{V_1, V_2\}$, e garante-se que $V \geq \frac{1}{2}\,\mathrm{OPT}$.

Listing 1. Pseudocódigo do Algoritmo 2-Aproximativo

```
Mochila2Aprox(valores[], pesos[], C):
  V1 <- max{valores[i] que respeita pesos[i] <= C}
  ordenar itens por (valores[i]/pesos[i]) desc.
  V2 <- 0; P <- 0
  para cada item em ordem:
    se P + pesos[item] <= C:
      V2 <- V2 + valores[item]
      P  <- P  + pesos[item]
  retornar max(V1, V2)
```

A prova de fator-2 segue dois casos:

- Se $V_1 \geq \frac{1}{2}\,\mathrm{OPT}$ então $\max\{V_1, V_2\} \geq V_1 \geq \frac{1}{2}\,\mathrm{OPT}$.

- Caso contrário, temos: $V_1 < \frac{1}{2}\,\mathrm{OPT}$. Seja S o conjunto escolhido pela heurística gulosa e k o primeiro item que não coube. Suponha que o peso total de S seja W, de modo que: $W + w_k > C$. No relaxamento fracionário, podemos inserir uma fração do item k, dada por:

$$\delta = \frac{C - W}{w_k},$$

o que rende valor adicional: $\delta\, v_k = (C - W)\left(\frac{v_k}{w_k}\right)$. Como a heurística gulosa é ótima para a mochila fracionária, temos:

$$\mathrm{OPT}_{\text{frac}} \geq V_2 + (C - W)\frac{v_k}{w_k} \leq V_2 + v_k,$$

e também: $\mathrm{OPT}_{\text{frac}} \geq \mathrm{OPT}$. Logo, concluímos:

$$V_2 + v_k \geq \mathrm{OPT} \quad \Rightarrow \quad \max\{V_2, v_k\} \geq \frac{1}{2}\,\mathrm{OPT}.$$

Como V_1 é o maior valor de item viável, temos: $V_1 \geq v_k$, portanto:

$$\max\{V_1, V_2\} \geq \max\{v_k, V_2\} \geq \frac{1}{2}\,\mathrm{OPT}.$$

Isso completa a prova de 2-aproximação.

A implementação segue uma estratégia simples e eficiente baseada em duas abordagens complementares. Os itens são inicialmente representados por pares `(densidade, índice)`, onde a densidade corresponde ao valor dividido pelo peso, e são ordenados em ordem decrescente. Para evitar divisão por zero durante o cálculo das densidades, é feita uma verificação explícita do peso de cada item. A estrutura utilizada — `std::vector<std::pair<double, int>>` — permite ordenação estável e eficiente com complexidade $O(n \log n)$.

A solução considera dois candidatos. O primeiro (Candidato 1) consiste no item de maior valor que não excede a capacidade da mochila, encontrado por busca linear sobre os dados originais. O segundo (Candidato 2) é obtido por uma varredura sequencial sobre os itens ordenados por densidade, acumulando valores enquanto a capacidade permite. A escolha final é o maior valor entre os dois candidatos, estratégia que assegura uma aproximação com garantia de 50% do ótimo. Esse método apresenta consumo de memória linear.

4. Análise Experimental e Discussão dos Resultados

Nesta seção, apresentamos a análise detalhada dos experimentos realizados com as três abordagens implementadas para o problema da mochila 0-1: o algoritmo exato Branch and Bound (BB), o esquema aproximativo FPTAS (Fully Polynomial-Time Approximation Scheme) parametrizado por ε, e o algoritmo heurístico 2-aproximativo desenvolvido neste trabalho.

Os testes foram realizados utilizando as instâncias disponibilizadas pelo professor, que abrangem uma variedade significativa de casos, desde problemas pequenos até instâncias maiores, com diferentes números de itens e capacidades.

As métricas analisadas incluem a qualidade da solução obtida, medida pela proximidade com o valor ótimo conhecido; o tempo de execução, registrado em milissegundos; e o uso de memória, medido em kilobytes. Todos os testes foram executados localmente, em uma máquina com processador Intel Core i5 e 16 GB de memória RAM. Todas as implementações foram feitas em C++.

A seguir, discutimos os resultados obtidos, comparando o desempenho dos algoritmos frente às instâncias testadas e destacando suas características em termos de eficiência e qualidade das soluções.

4.1. Qualidade das Soluções

Em termos de qualidade, o algoritmo Branch and Bound serviu como referência absoluta, fornecendo sempre a solução ótima para todas as instâncias testadas. Mesmo nas instâncias de maior porte, como a `knapPI_3_500_1000_1` com 500 itens, o algoritmo obteve resultados corretos, embora com tempos de execução elevados. Esses resultados confirmam a robustez e exatidão da implementação.

O FPTAS, configurado com $\varepsilon = 0{,}5$ (50% de tolerância), apresentou qualidade consistente, alinhada com as garantias teóricas do método. A média de aproximação ao ótimo ficou em torno de 97%, com erros máximos mais pronunciados em instâncias pequenas. Para casos maiores, como `knapPI_1_500_1000_1`, o erro percentual foi

inferior a 0,1%, indicando que o impacto do arredondamento diminui conforme aumenta o número de itens, devido à redução relativa do erro de escala.

O algoritmo 2-aproximativo apresentou maior variabilidade na qualidade das soluções. Em diversas instâncias, o erro percentual ficou abaixo de 2%, demonstrando bom desempenho para conjuntos maiores. Entretanto, em algumas instâncias pequenas, como `f4_l-d_kp_4_11`, o erro chegou a 30,44%, o que evidencia a limitação do método para casos onde o item de maior valor isolado não representa bem a solução combinada ótima. De modo geral, o algoritmo 2-aproximativo fornece soluções próximas do ótimo com baixo custo computacional, mas pode apresentar resultados significativamente inferiores em perfis de dados específicos.

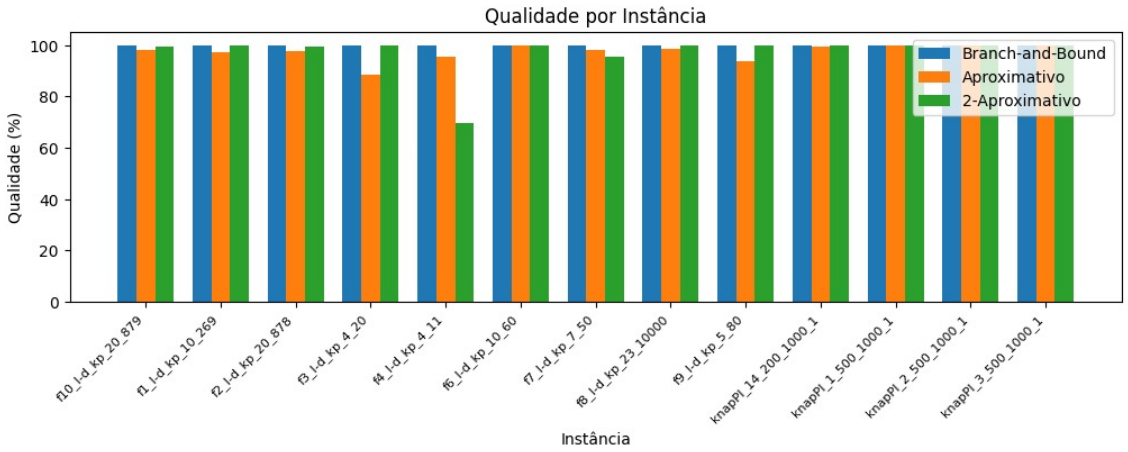

Figure 1. Comparação da qualidade da solução obtida (percentual do valor ótimo).

4.2. Tempos de Execução

Os tempos de execução obtidos nas instâncias testadas apresentam variações significativas entre os algoritmos, conforme detalhado a seguir.

O algoritmo Branch and Bound (BB) mostrou tempos que variam desde valores muito baixos em instâncias pequenas (por exemplo, 0.013 ms em `f9_l-d_kp_5_80`) até valores elevados em instâncias maiores, chegando a 195334 ms (aproximadamente 195 segundos) em `knapPI_1_500_1000_1` e 262137 ms (cerca de 262 segundos) em `knapPI_3_500_1000_1`. Esse crescimento exponencial está alinhado com a natureza combinatória do problema e a necessidade de explorar extensivamente o espaço de soluções para garantir a otimalidade.

O algoritmo FPTAS apresentou tempos consistentemente baixos, variando de 0.018 ms (`f3_l-d_kp_4_20`) a cerca de 1.7 segundos (1759.06 ms em `knapPI_3_500_1000_1`). Em comparação com o Branch and Bound, o FPTAS manteve um tempo de execução muito inferior nas instâncias maiores, sendo até cerca de 150 vezes mais rápido no caso `knapPI_3_500_1000_1`.

O algoritmo 2-aproximativo exibiu os menores tempos de execução em todos os casos testados, com valores entre 0.008 ms (`f4_l-d_kp_4_11`) e 0.292 ms (`knapPI_1_500_1000_1`), mantendo desempenho estável e mínimo mesmo para

instâncias maiores. Essa rapidez decorre da simplicidade da heurística utilizada, que evita cálculos complexos ou buscas extensivas.

Em resumo, para instâncias pequenas (n e W reduzidos), os três algoritmos apresentam tempos muito baixos, com diferenças marginais. À medida que a complexidade cresce, o Branch and Bound sofre aumento exponencial no tempo, tornando-se inviável para casos grandes, enquanto o FPTAS e o 2-aproximativo mantêm tempos baixos e escaláveis, com o 2-aproximativo sendo o mais eficiente.

Os dados reforçam o trade-off clássico entre qualidade e tempo: o Branch and Bound garante solução ótima ao custo de alto tempo em instâncias grandes, o FPTAS oferece solução próxima do ótimo em tempo polinomial, e o 2-aproximativo prioriza tempo mínimo com garantias teóricas menos rígidas.

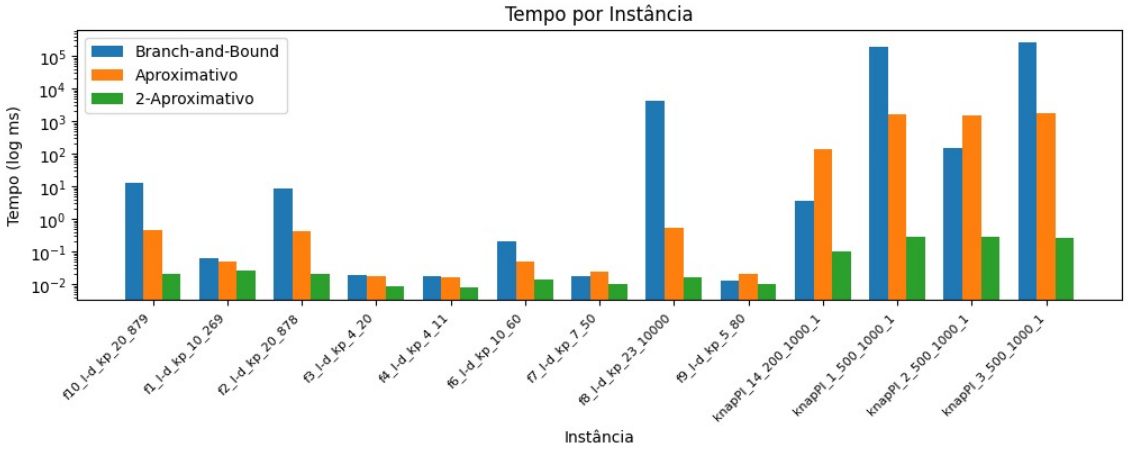

Figure 2. Comparação do tempo de execução (ms) em escala logarítmica.

4.3. Uso de Memória

O algoritmo Branch and Bound apresentou consumo de memória significativamente maior em comparação às outras abordagens. Em instâncias pequenas, o pico de memória situou-se na faixa de aproximadamente 3,4 MB, porém, para problemas maiores, como o conjunto knapPI_3_500_1000_1, o uso ultrapassou 3 GB. Esse comportamento decorre da estrutura da busca em árvore, que mantém múltiplos nós simultaneamente na memória, resultando em crescimento exponencial do uso conforme o aumento da dimensão do problema.

A abordagem FPTAS exibiu um padrão intermediário de consumo. Nas instâncias menores, o uso de memória ficou próximo de 3,3 MB, similar ao algoritmo 2-aproximativo, mas em instâncias maiores, como knapPI_3_500_1000_1, o consumo alcançou aproximadamente 547 MB. Esse incremento está associado à necessidade de armazenamento das tabelas de programação dinâmica, cujo tamanho depende tanto do número de itens quanto do fator de arredondamento adotado para garantir a precisão da solução.

Por sua vez, o algoritmo 2-aproximativo manteve um uso de memória consistentemente baixo, em torno de 3,3 MB mesmo para as maiores instâncias testadas. Essa estabilidade é explicada pela simplicidade da heurística, que utiliza apenas vetores lineares

para armazenar valores, pesos e densidades, sem a necessidade de estruturas complexas ou tabelas extensas. Essa característica torna essa abordagem especialmente adequada para ambientes com restrição severa de memória.

Em suma, o consumo de memória cresce de forma acentuada no Branch and Bound devido à complexidade estrutural da busca, apresenta crescimento moderado no FPTAS condicionado à precisão desejada e mantém-se baixo e estável no algoritmo 2-aproximativo pela simplicidade da estratégia adotada.

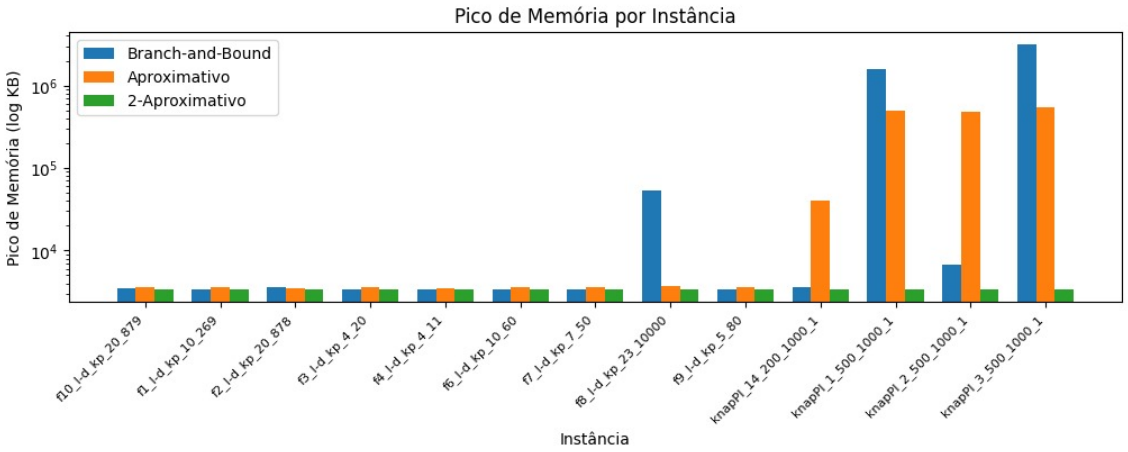

Figure 3. Comparação entre os picos de memória (KB) em escala logarítmica.

4.4. Comparação Geral e Aplicabilidade

Cada abordagem se destacou em aspectos diferentes. O **Branch and Bound** é adequado quando a solução ótima é necessária, mas apresenta alto custo de tempo e memória em instâncias grandes. O **FPTAS**, com $\varepsilon = 0{,}5$, produziu soluções próximas do ótimo com menor custo computacional, sendo viável para instâncias médias ou quando se admite pequeno erro. Já o **2-aproximativo** teve o melhor desempenho em tempo e memória, com soluções aceitáveis, sendo indicado para cenários com restrições de recursos ou quando se busca uma resposta rápida.

Critério	**Algoritmo Recomendado**
Solução ótima	Branch and Bound
Boa aproximação com desempenho	FPTAS (ajuste por ε)
Execução rápida e leve	2-aproximativo
Estimativa inicial	2-aproximativo / FPTAS

5. Conclusões

Os experimentos realizados demonstram que cada abordagem apresenta vantagens e limitações claras, que devem guiar sua escolha conforme o contexto de aplicação. O Branch and Bound garante exatidão, mas tem escalabilidade limitada; o FPTAS oferece controle explícito da aproximação e melhor escalabilidade, mas ainda demanda recursos consideráveis; o 2-aproximativo prioriza velocidade e baixo consumo de memória, com compromissos em qualidade.

A implementação e avaliação conjunta desses algoritmos permitem oferecer alternativas para diferentes necessidades, tornando possível ajustar a solução de acordo com requisitos de tempo, memória e precisão. Este estudo evidencia a importância de considerar múltiplas estratégias para problemas NP-difíceis, explorando o compromisso entre exatidão e eficiência.

6. References

Vimieiro, R. (2025). Slides da disciplina Algoritmos 2. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais (UFMG).

Wikipedia contributors. (2025). Knapsack problem. In Wikipedia, The Free Encyclopedia. Retrieved from
https://en.wikipedia.org/wiki/Knapsack_problem