Trabalho Prático 2 - Algoritmos Aproximativos

Bernardo A. Miranda[1], **Bernardo V. C. Oliveira**[1], **Lucas A. S. Costa**[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Av. Antônio Carlos, 6627 – Pampulha – 31270-901 – Belo Horizonte – MG – Brazil

Resumo. *Este trabalho investiga soluções exatas e aproximadas para o Problema da Mochila 0-1, implementando algoritmos de Branch-and-Bound com busca best-first, FPTAS com precisão* $\epsilon = 0.5$ *e um algoritmo aproximativo guloso. São discutidas as decisões de implementação, estruturas de dados utilizadas e os experimentos realizados, visando comparar tempo de execução, uso de memória e qualidade das soluções encontradas.*

1. Introdução

O Problema da Mochila Binária (0-1 Knapsack Problem) é um problema clássico NP-difícil. Dada uma mochila com capacidade limitada e um conjunto de itens com pesos e valores, o objetivo é determinar o subconjunto que maximiza o valor total sem exceder a capacidade. Este trabalho propõe implementar e comparar três abordagens distintas, visando analisar suas vantagens, limitações e o comportamento prático em diferentes instâncias.

2. Descrição Detalhada dos Algoritmos

2.1. Branch-and-Bound com Busca Best-First

O Branch-and-Bound explora o espaço de soluções através de uma árvore binária de decisões, onde cada nó representa a escolha de incluir ou não o próximo item na mochila. A ordem de exploração que escolhemos utilizar foi determinada por uma estratégia de busca *best-first*, que utiliza uma fila de prioridade para sempre expandir o nó mais promissor, com base em uma estimativa de *upper bound* do valor total que ainda pode ser alcançado a partir daquele ponto.

A eficiência do método depende fortemente da capacidade de podar subárvores inviáveis. Para isso, comparamos o bound de um nó com o valor da melhor solução inteira encontrada até o momento (*lower bound*). Se o bound for inferior ou igual, o nó é descartado junto com todos os seus descendentes, pois não pode conter uma solução melhor.

2.1.1. Análise de Trade-off das Funções de Bound

O custo e a precisão do cálculo do bound afetam diretamente o desempenho do algoritmo. Um bound mais justo tende a eliminar mais nós, mas pode ser computacionalmente mais caro. Por outro lado, um bound mais simples é rápido, porém pode permitir a exploração de mais nós desnecessários. Para investigar esse trade-off, implementamos duas funções de bound com diferentes níveis de complexidade.

Bound por Relaxação Linear (Complexidade $O(k)$) Essa abordagem calcula um bound justo ao resolver a relaxação fracionária do subproblema. A partir do nó atual, adiciona-se os próximos k itens inteiros que ainda cabem na mochila, preenchendo o espaço restante com uma fração do próximo item. Essa estimativa é mais próxima do valor ótimo, permitindo maior eficiência na poda.

Bound por Melhor Razão Local (Complexidade $O(1)$) Essa abordagem fornece uma estimativa mais rápida, mas menos precisa. Assume-se que todo o espaço restante na mochila pode ser preenchido com cópias do item disponível com a melhor razão valor/peso, resultando em um bound otimista e computacionalmente leve.

Optamos pela estratégia *best-first search*, priorizando nós com maior bound, implementada por meio de uma fila de prioridade (min-heap invertida), garantindo complexidade $\mathcal{O}(\log n)$ nas operações de inserção e remoção. Para representar os nós, usamos objetos Python com informações do estado parcial: índice do item atual, lucro acumulado, peso acumulado e bound.

2.2. FPTAS com $\epsilon = 0.5$

O Fully Polynomial-Time Approximation Scheme (FPTAS) transforma o problema original em uma versão aproximada escalonando os valores dos itens de acordo com a precisão ϵ. Cada valor é reduzido proporcionalmente, garantindo que a solução encontrada difira do ótimo em no máximo um fator de $(1 - \epsilon)$. Utilizamos programação dinâmica sobre os valores escalonados: seja P_{max} o maior valor entre os itens, definimos $K = \epsilon P_{max}/n$, escalonando os lucros para $\lfloor p_i/K \rfloor$.

Implementamos a tabela de programação dinâmica utilizando uma lista, porém notamos que essa estrutura de dados não era a ideal, pois não aproveita uma característica importante: o fato de que as entradas são intrínsicamente esparsas - muitos valores de lucros intermediários nunca são alcançados. Por isso, desenvolvemos uma versão utilizando dicionários, que se mostrou extremamente mais eficiente no gerenciamento de memória, uma vez evita alocar espaço para entradas desnecessárias. A complexidade do FPTAS é $\mathcal{O}\left(\frac{n^3}{\epsilon}\right)$.

2.3. Algoritmo Aproximativo Guloso e Prova de Aproximação

Implementamos o algoritmo guloso clássico, que ordena os itens por densidade valor/peso (p_i/s_i) e os insere na mochila até atingir a capacidade. No entanto, este algoritmo puro pode ter desempenho arbitrariamente ruim em casos específicos. Para melhorar, usamos a abordagem do *ModifiedGreedy*, que seleciona entre a solução gulosa e o item individual de maior valor, garantindo uma 2-aproximação.

Demonstração simplificada: Seja OPT o valor ótimo. Se $p_1 + \ldots + p_{k-1} \geq OPT/2$, a solução gulosa é 2-aproximada. Caso contrário, o item k isolado (ou sua fração que ainda cabe) garante pelo menos $OPT/2$, pois a soma dos itens até k cobre pelo menos metade do ótimo.

3. Metodologia Experimental

Executamos os três algoritmos em instâncias provenientes de dois repositórios amplamente utilizados na literatura para avaliação do problema da mochila 0-1:

- `http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/`
- `https://www.kaggle.com/datasets/sc0v1n0/large-scale-01-knapsackproblems`

Os experimentos foram conduzidos em um **Notebook Lenovo com processador AMD Ryzen 5 3500U, 12 GB de RAM**, rodando **Linux Mint 22.1 "Xia"**. Os algoritmos foram implementados em **Python 3.12.3**. Seguindo as recomendações do enunciado.

Cada instância foi limitada a um tempo máximo de execução de 30 minutos (como solicitado no enunciado), para manter a viabilidade do trabalho. As principais métricas coletadas em todos os experimentos incluíram:

- **Tempo de execução** total em segundos;
- **Memória máxima utilizada** durante a execução.
- **Dados específicos de instância:** Usamos fatores como o valor da solução encontrada, o tamanho da mochila etc para medir qualitativamente as soluções encontradas.

Para garantir a consistência dos resultados, cada instância foi executada três vezes consecutivas, e os valores médios das métricas foram considerados. As medições de tempo de execução e memória foram realizadas exclusivamente com bibliotecas padrão do Python, como `time` (para mensuração do tempo) e o módulo `tracemalloc` (para obtenção do pico de uso de memória), assim como solicitado no enunciado.

4. Resultados

4.1. Desempenho em Tempo

A Figura 1 apresenta a distribuição do tempo de execução dos algoritmos Branch-and-Bound, FPTAS e Greedy. O boxplot, em escala logarítmica, evidencia diferenças na tendência central e na variação entre os métodos, permitindo observar que o Greedy possui tempos consistentemente mais baixos, enquanto o FPTAS apresenta maior dispersão em algumas instâncias.

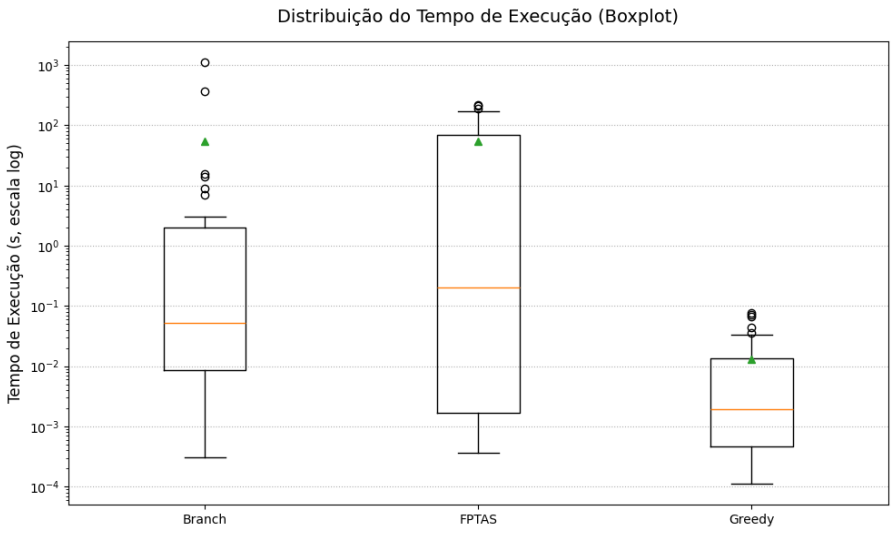

Figure 1. Distribuição do tempo de execução entre os algoritmos Branch-and-Bound, FPTAS e Greedy.

A Figura 2 ilustra a evolução do tempo de execução em função do número de itens nas instâncias de teste. Esse gráfico destaca como cada algoritmo escala: nota-se que o FPTAS é mais sensível ao aumento do número de itens, enquanto o Greedy mantém tempo de execução baixo mesmo em problemas maiores.

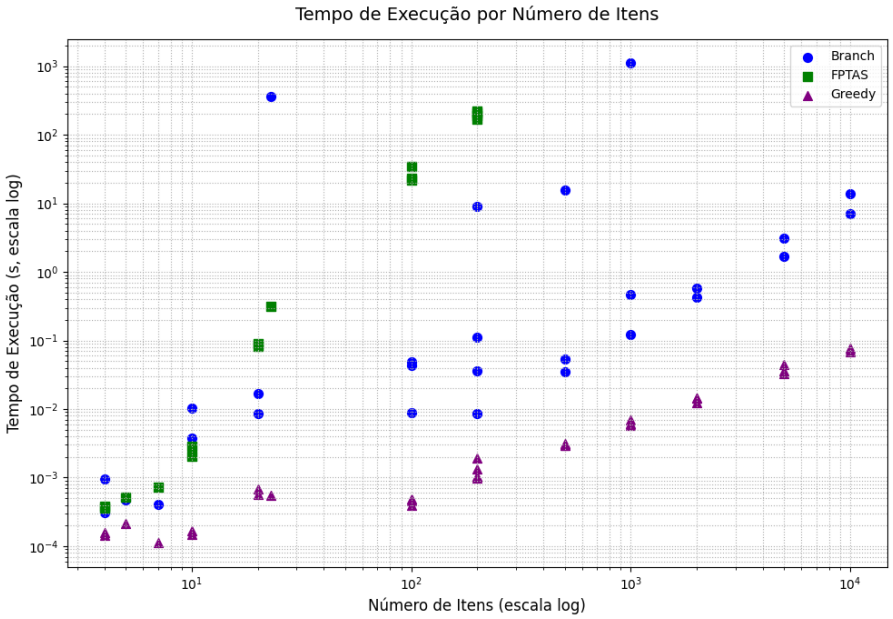

Figure 2. Tempo de execução dos algoritmos em função do número de itens, destacando o impacto da escalabilidade.

4.2. Uso de Memória

A Figura 3 mostra como o uso de memória pico evolui em função do número de itens para os algoritmos Branch-and-Bound, FPTAS e Greedy. É possível observar que o método Greedy apresenta consumo de memória praticamente constante e muito baixo, mesmo em instâncias grandes, evidenciando sua leveza em termos de recursos. Já o Branch-and-Bound apresenta crescimento acentuado de memória conforme o tamanho do problema aumenta, atingindo valores muito altos em instâncias com milhares de itens. O FPTAS, por sua vez, mostra consumo intermediário: menor que Branch em muitos casos, mas ainda significativamente superior ao Greedy, especialmente em instâncias de maior porte.

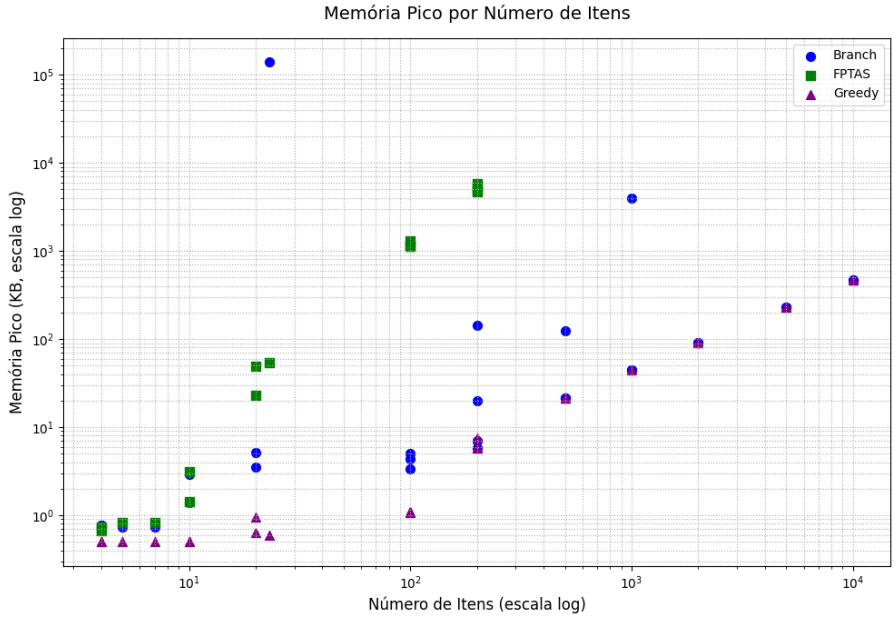

Figure 3. Uso de memória (em KB) dos algoritmos em função do número de itens, em escala log-log.

4.3. Qualidade da Solução

A Figura 4 apresenta o erro relativo dos algoritmos aproximativos FPTAS e Greedy em relação ao valor ótimo. Nota-se que o FPTAS mantém erro consistentemente baixo, abaixo de 1/100 na maioria das instâncias, comprovando sua garantia de aproximação. Já o Greedy demonstra variação muito maior, com erros que podem ultrapassar 30/100 em algumas instâncias, especialmente nas de menor tamanho, indicando menor confiabilidade para fornecer soluções próximas do ótimo.

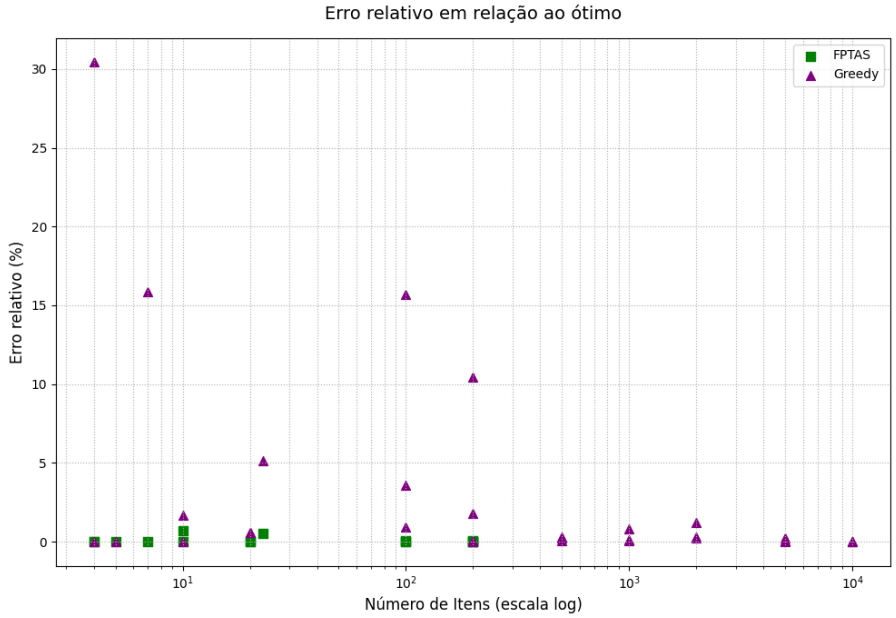

Figure 4. Erro relativo dos algoritmos aproximativos (FPTAS e Greedy) em relação ao valor ótimo.

4.4. Instâncias não Solucionadas

A Figura 5 apresenta o mapa de calor das instâncias que ultrapassaram o tempo de execução definido de 30 minutos. Com base nela, é possível observar que o algoritmo implementado com base no FPTA sofre bastante à medida que as instâncias crescem de tamanho, enquanto isso não é refletido de maneira tão acentuada no algoritmo de branch and bound e tampouco na abordagem gulosa como era o esperado.

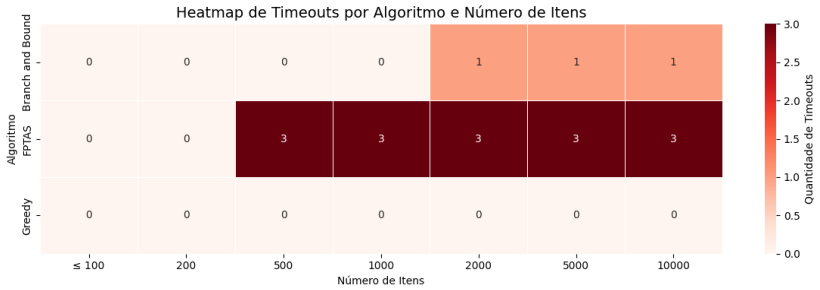

Figure 5. Numero de Timeouts ocorridos por tamanho de instância

4.5. Análise dos Bounds: Linear vs. Constante

Durante a execução do algoritmo de Branch-and-Bound para o Problema da Mochila, dois tipos de *bound* com relaxação linear foram testados:um *bound* com custo linear e o outro *bound* com custo constante. O primeiro é mais preciso, pois calcula o valor máximo possível levando em consideração a possibilidade de adicionar frações dos itens após preencher a mochila com os melhores itens inteiros, enquanto o segundo é mais simples, baseando-se apenas no item com o maior custo-benefício (valor/peso). Embora

o *bound* linear demande mais tempo para cálculo, seu impacto na eficiência do algoritmo é significativo.

4.5.1. Bound Linear

O *bound* linear calcula uma estimativa do valor total possível da solução, considerando as frações dos itens para preencher a mochila após não ser mais possível acrescentar itens inteiros. Esse cálculo é feito de maneira linear no número de itens, ou seja, seu custo computacional é proporcional ao número de itens. Apesar de seu custo mais alto, esse *bound* se mostrou muito mais eficaz, pois permite uma poda mais eficiente da árvore de busca, resultando em um número significativamente menor de nós explorados, o que diminui o uso de memória e o tempo de execução.

4.5.2. Bound Constante

Por outro lado, o *bound* constante, que é calculado em tempo $O(1)$ através da simples escolha do item de maior valor/peso, tem a vantagem de ser muito rápido. No entanto, essa simplicidade vem com um custo: a poda das subárvores ocorre de forma mais lenta, já que a estimativa do valor máximo possível é menos precisa. Isso resulta em uma expansão muito maior da árvore de busca, com muitos nós desnecessários sendo explorados.

4.5.3. Resultados Experimentais

Os resultados experimentais, evidenciados pelas imagens a seguir, mostram claramente o impacto de cada abordagem na execução do algoritmo. Embora o *bound* linear leve mais tempo para ser calculado devido à sua dependência do número de itens, ele é substancialmente mais eficaz em termos de redução do número de nós explorados e do tamanho da fila de nós.

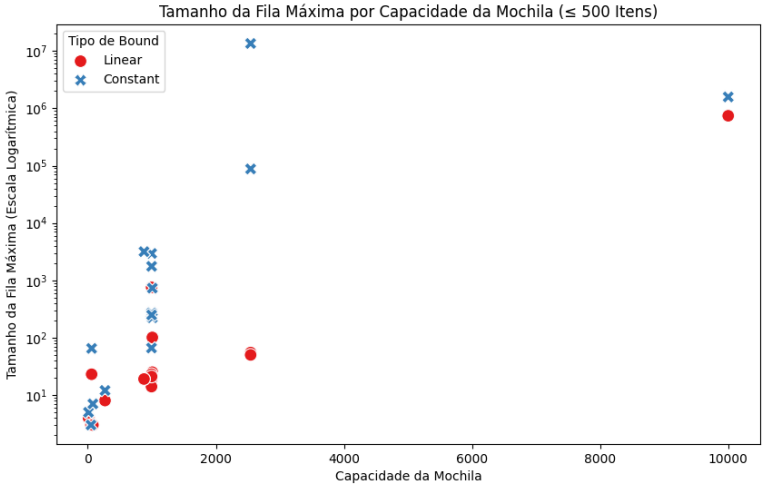

Figure 6. Tamanho máximo da fila de prioridade

Como mostrado na Figura 6, o *bound* linear permite uma poda mais agressiva, resultando em uma fila muito menor em comparação com o *bound* constante. A fila de nós, no caso do *bound* constante, cresce rapidamente, já que a estimativa de valor máximo possível não contribui significativamente para a poda.

Além disso, a Figura abaixo ilustra o número total de nós explorados durante a execução do algoritmo:

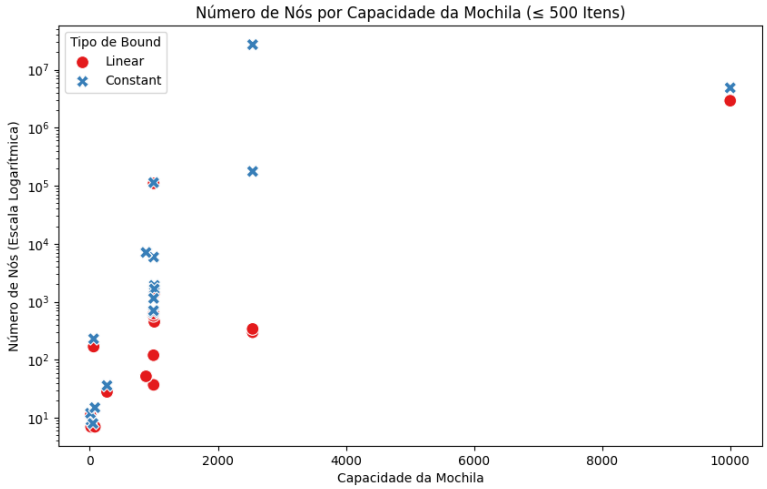

Figure 7. Numero de Nós criados

Na Figura 7, é possível observar que o número de nós criados é significativamente menor quando se utiliza o *bound* linear, evidenciando que a poda ocorre de forma mais eficiente, o que resulta em um desempenho melhor para o algoritmo como um todo.
Isso reflete tanto na memória gasta, pois quanto mais nós são armazenados na fila e criados, mais memória gasta e mais lento se torna o algoritmo. Sendo assim optamos por utilizar o Branch And Bound com o primeiro bound apresentado.

5. Discussão

Os resultados evidenciaram diferenças claras entre as abordagens avaliadas:

- **Branch-and-Bound (B&B)** apresentou um desempenho excelente em termos de qualidade das soluções, uma vez que sempre fornece resultados exatos, atingindo o valor ótimo em todas as instâncias que conseguiu resolver. Ao compará-lo com o FPTAS, observou-se que, apesar de um ser um algoritmo exponencial e o outro polinomial, o B&B foi capaz de resolver instâncias maiores que o FPTAS não conseguiu dentro do prazo estipulado. Além disso, nos casos em que ambos os algoritmos produziram resultados, o tempo de execução do B&B foi, em geral, menor, especialmente em instâncias maiores.
- **FPTAS**, por outro lado, foi uma decepção. Não conseguiu cumprir as expectativas que foram inicialmente colocadas, e, com base nas análises realizadas, pode-se concluir que ele é mais uma promessa do que uma solução prática. Apesar do bom desempenho assintótico, na prática ele não oferece benefícios em termos de

tempo, memória ou qualidade das soluções. O FPTAS não apenas ficou atrás do B&B, como também não garante uma resposta exata, apenas assegura que a solução será suficientemente próxima do ótimo (com uma precisão de 50%, no caso). Se for necessário aumentar a qualidade da solução, a complexidade do algoritmo se eleva, resultando em maior tempo de execução e uso de memória. Isso torna o FPTAS, em muitos casos, menos eficiente que o B&B, sem oferecer a garantia de uma solução exata. Embora o algoritmo mantenha uma precisão consistente, o trade-off entre flexibilidade e desempenho é evidente, o que torna sua aplicabilidade questionável em muitos cenários.

- **Heurística Gulosa** destacou-se pela rapidez de execução, conseguindo resolver as maiores instâncias de forma ágil. No entanto, apresentou erros significativos em instâncias que envolviam itens com grande variação na relação valor/peso, o que revela suas limitações em situações onde a precisão é fundamental. Apesar disso, em cenários nos quais o tempo de resposta é prioritário, a heurística gulosa pode ser uma escolha vantajosa devido à sua eficiência.

Em suma, a escolha do algoritmo depende de um balanço estratégico. O `B&B` firma-se como a melhor opção quando a otimalidade é um requisito inegociável e o risco de um tempo de execução exponencial em instâncias bem específicas é tolerável. Em contraste, o `FPTAS` torna-se relevante em sistemas onde uma garantia de tempo de execução polinomial é mandatória para evitar falhas, aceitando-se uma solução aproximada, mas com qualidade controlável. Por fim, a heurística gulosa serve como uma eficiente linha de base, ideal para obter soluções rápidas e de qualidade razoável com baixo custo computacional.

6. Conclusão

Este trabalho apresentou um estudo experimental detalhado sobre o problema da mochila 0-1, comparando algoritmos exatos e aproximativos em termos de precisão, tempo de execução e uso de memória. Foram implementados três algoritmos distintos: Branch-and-Bound, como método exato; FPTAS, como abordagem de aproximação com garantias teóricas; e uma heurística gulosa baseada na densidade valor/peso.

As implementações foram avaliadas em instâncias clássicas disponíveis em repositórios reconhecidos, com diferentes tamanhos e características, permitindo observar empiricamente como cada algoritmo se comporta em cenários variados. As execuções foram conduzidas em ambiente controlado, coletando métricas detalhadas como tempo de execução, memória consumida, número de nós gerados e qualidade da solução em relação ao ótimo conhecido. Os resultados confirmaram que o algoritmo Branch-and-Bound, ao contrário do esperado, se mostrou extremamente eficiente devido às suas podas, destacando-se como a melhor opção quando se busca obter melhores resultados. Por outro lado, o desempenho do FPTAS apresentou um ponto negativo significativo, pois, com um valor de episilon ligeiramente baixo, o algoritmo se mostrou muito ineficiente em todos os quesitos analisados. Foi possível perceber na prática que não basta apenas olhar para o custo teórico dos algoritmos, é necessário entender como eles se comportam na prática. Além disso, notamos que, apesar de o Branch-and-Bound apresentar ótimos desempenhos, quando testamos diferentes formas de calcular o bound, o desempenho do algoritmo variou drasticamente, levando a resultados completamente diferentes. Isso evidencia que deve ser dada uma atenção especial a essa escolha ao utilizar esse tipo de

algoritmo.

De forma geral, este estudo ilustra as principais trocas entre qualidade de solução e eficiência computacional nas abordagens para o problema da mochila 0-1, além de evidenciar a importância da escolha do algoritmo de acordo com as características e restrições do problema real a ser resolvido.