Benchmark de algoritmos para o problema da mochila Papers and Abstracts

Haniel B. Ribeiro[1], **Pedro L. Alzamora**[1]

[1]Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG)

{haniel.botelho,pedroloures}@dcc.ufmg.br

Resumo. *Este artigo realiza um benchmark de três algoritmos para o problema da mochila: o exato Branch and Bound (BB), o aproximativo FPTAS e um Guloso aprimorado. O estudo avalia o trade-off entre tempo de execução e qualidade da solução. Os resultados mostram que o BB, embora garanta a solução ótima, é exponencialmente lento para instâncias grandes. O FPTAS demonstrou ser impraticável para larga escala, falhando por excesso de consumo de memória. Em contrapartida, o algoritmo Guloso se destacou por ser extremamente rápido e por encontrar soluções de alta qualidade, com resultados frequentemente superiores a 95% do valor ótimo. A conclusão é que a abordagem gulosa foi a mais eficiente e vantajosa na prática para os casos testados.*

Abstract. *This paper presents a benchmark of three algorithms for the knapsack problem: the exact Branch and Bound (BB), the approximate Fully Polynomial-Time Approximation Scheme (FPTAS), and an improved Greedy algorithm. The study evaluates the trade-off between execution time and solution quality. Results show that while BB guarantees the optimal solution, it is exponentially slow for large instances. The FPTAS proved to be impractical for large-scale problems, failing due to excessive memory consumption. In contrast, the Greedy algorithm stood out for being extremely fast and finding high-quality solutions, often exceeding 95% of the optimal value. The conclusion is that the greedy approach was the most efficient and advantageous in practice for the tested cases.*

1. Introdução

O presente trabalho busca avaliar o desempenho de diferentes algoritmos para o problema da mochila binário, tanto em termos de custo computacional quanto na qualidade da resposta. O propósito central deste estudo é analisar o compromisso (trade-off) entre a eficiência de um algoritmo e a otimalidade da solução que ele produz. Busca-se, portanto, investigar como as características de diferentes instâncias do problema influenciam essa relação, a fim de determinar sob quais condições uma abordagem aproximativa e rápida pode ser mais vantajosa que uma solução exata, porém custosa.

Portanto, contextualizemos o problema da **Mochila Binário** (Knapsack 0/1 Problem). Sejam o conjunto $\{I_1, I_2, \ldots, I_n\}$ e um inteiro C a instância do problema. Onde C representa a capacidade máxima da mochila, e cada I_i é uma tupla (v_i, w_i), sendo v_i e w_i o valor e o peso, respectivamente. Dado uma entrada o objetivo é encontrar o conjunto $S \subset I$ tal que $\sum_{i \in S} w_i \leq C$ e maximize $\sum_{i \in S} v_i$.

Tendo em vista que esse é um problema clássico na história da ciência da computação, e que prova de que o Problema da Mochila é NP-Difícil é bem estabelecida na literatura e será omitida neste trabalho. Dado o custo computacional de uma abordagem por força bruta, estratégias mais eficientes tornam-se necessárias. Portanto as soluções propostas são: a abordagem por branch-and-bound e a abordagem conhecida como esquema de aproximação de tempo totalmente polinomial (FPTAS).

A ideia geral por trás do branch-and-bound é fazer a busca exaustiva, mas fazendo eliminando caminhos infrutíferos e priorizando caminhos mais promissores. Por outro lado, a abordagem FPTAS busca uma solução em tempo polinomial por um fator de aproximação passado por parâmetro. Seguindo esse pensamento, o artigo se desenvolverá nas seguintes seções: descrição dos algoritmos implementados; análise do resultado experimental; por fim, a conclusão deste trabalho.

2. Algoritmos utilizados

2.1. Algoritmo Branch and Bound

A implementação do algoritmo B&B para o Problema da Mochila 0/1 segue os princípios de ramificação e limitação, com otimizações para eficiência:

2.1.1. Escolha da Estimativa de Custo (Função de Limite - Bounding Function)

A eficácia do algoritmo B&B depende criticamente da qualidade da função de limite superior. Uma função de limite mais ”apertada” (que estima o lucro máximo de forma mais precisa) permite uma poda mais agressiva do espaço de busca, reduzindo o tempo de execução.

Nesta implementação, a estimativa de custo utilizada é baseada na **relaxação da mochila fracionária**. Para um determinado nó na árvore de busca (que representa uma seleção parcial de itens), o limite superior é calculado da seguinte forma:

1. Soma-se o lucro dos itens já incluídos na mochila para aquele nó.
2. Para os itens restantes (ainda não decididos), eles são considerados em ordem decrescente de sua **razão lucro/peso (p_i/w_i)**.
3. Estes itens são adicionados ”fracionariamente” até que a capacidade restante da mochila seja preenchida. Se um item inteiro não couber, uma fração dele é adicionada para usar a capacidade remanescente, e seu lucro proporcional é somado.

Esta abordagem fornece um limite superior válido porque a solução fracionária sempre resultará em um lucro igual ou maior do que a solução ótima binária para o mesmo subproblema. A escolha da relaxação fracionária é vantajosa por ser relativamente simples e rápida de calcular, ao mesmo tempo em que oferece um limite razoavelmente justo.

2.1.2. Estruturas de Dados Utilizadas e o Porquê

- **Itens:** Cada item é representado como uma tupla `(lucro, peso, indice_original)`. O `indice_original` é crucial para reconstruir a solução final na ordem original dos itens, após a ordenação interna do algoritmo.

- **Nós da Árvore de Busca:** Embora não haja uma estrutura de ”nó” explícita como um objeto, cada estado na pilha do algoritmo iterativo representa um nó. Este estado inclui: `(nível, lucro_atual, peso_atual, seleção_atual)`.
  - `nível`: O índice do item que está sendo considerado atualmente.
  - `lucro_atual`: O lucro total dos itens selecionados até o momento nesta ramificação.
  - `peso_atual`: O peso total dos itens selecionados até o momento nesta ramificação.
  - `seleção_atual`: Uma lista booleana (ou similar) que registra quais itens foram incluídos na mochila para a ramificação atual, usando seus índices originais.
- **Pilha Explícita para Busca (Stack):** Para a implementação do algoritmo de busca, foi utilizada uma lista Python como uma pilha explícita. Esta escolha foi feita para simular uma **Busca em Profundidade (Depth-First Search - DFS) iterativa**, evitando os limites de recursão padrão do Python (`sys.setrecursionlimit`) que seriam um problema para instâncias com um grande número de itens (até 10.000). A DFS iterativa também é mais eficiente em termos de uso de memória em comparação com uma Busca em Largura (BFS) que exigiria uma fila ou fila de prioridade para armazenar um grande número de nós abertos.

2.1.3. Escolha de Best-First ou Depth-First

A implementação utiliza uma estratégia de **Busca em Profundidade (Depth-First Search - DFS) iterativa**. As razões para esta escolha são:

- **Eficiência de Memória:** A DFS requer significativamente menos memória do que a Best-First Search (que usa uma fila de prioridade para armazenar todos os nós ”promissores” abertos). Para problemas com um grande número de itens, a BFS pode rapidamente esgotar a memória disponível.
- **Encontrar Soluções Viáveis Rapidamente:** A DFS tende a encontrar uma solução viável (mesmo que não ótima) mais rapidamente, pois explora um caminho até o fim. Uma vez que uma solução viável é encontrada, o `max_profit` (melhor lucro global encontrado até o momento) é atualizado, permitindo uma poda mais agressiva de outras ramificações.

2.1.4. Detalhes de Implementação e Otimizações

- **Ordenação dos Itens:** Os itens são pré-ordenados em ordem decrescente de sua razão lucro/peso (p_i/w_i). Esta heurística é crucial para a função de limite, pois permite que os itens mais ”valiosos” por unidade de peso sejam considerados primeiro, levando a limites superiores mais apertados e, consequentemente, a uma poda mais eficaz.
- **Inicialização do `max_profit` (Heurística Gulosa):** Antes de iniciar o processo de B&B, um valor inicial para `max_profit` (o melhor lucro encontrado até o momento) é estabelecido usando uma heurística gulosa simples. Esta heurística

seleciona itens na ordem de sua razão lucro/peso até que a mochila esteja cheia. Embora não garanta a otimalidade, ela fornece um bom limite inferior inicial, permitindo que o algoritmo B&B pode ramificações de forma mais eficiente desde o início.

- **Condições de Poda (Pruning):** Dois critérios de poda são aplicados em cada nó:

    1. **Poda por Peso:** Se o `peso_atual` exceder a `capacidade` da mochila, a ramificação é inválida e é imediatamente podada.
    2. **Poda por Limite Superior:** Se o `limite_superior` calculado para o nó atual for menor ou igual ao `max_profit` global já encontrado, essa ramificação não pode levar a uma solução melhor e é podada.

- **Controle de Tempo:** Um mecanismo de controle de tempo é implementado para garantir que o algoritmo não exceda um limite de tempo pré-definido (e.g., 30 minutos). O algoritmo verifica o tempo decorrido periodicamente (a cada 10.000 nós explorados) e termina se o limite for atingido, retornando a melhor solução encontrada até aquele momento.
- **Indicação de Progresso (`tqdm`):** A biblioteca `tqdm` é utilizada para fornecer uma barra de progresso visual no terminal. Esta barra indica o progresso em termos do "nível" de itens que o algoritmo está processando, dando uma estimativa visual de que o programa está em execução, especialmente para instâncias maiores.

2.2. FPTAS

Como dito anteriormente, em algumas ocasiões, deseja-se obter uma solução que seja X% pior que a ótima. Existem algoritmos onde esse fator *X* se torna um parâmetro do algoritmo. Tais algoritmos são conhecidos como esquemas de ***aproximação de tempo polinomial*** **(PTAS)**. Onde o tempo de execução desses algoritmos dependem tanto do tamanho da entrada *n* quanto do parâmetro de qualidade ε . Essa dependência de ε pode ser, inclusive, exponencial. Quando ela for polinomial, o algoritmo é chamado de fully polynomial-time approximation scheme (FPTAS), esse será o algoritmo abordado nesta seção.

O problema da mochila binário admite uma solução pseudo-polinomial com tempo $O(nW)$, se o peso W não for grande. Podemos utilizar programação dinâmica **(DP)** para resolver o problema, onde respondemos a pergunta: qual o valor máximo que podemos obter com os k primeiros itens dentro do peso admitido? Essa é a base para o nosso FPTAS, podemos reformular o problema em termos do *valor* para encontrar uma solução com tempo $O(nV)$, onde $V = \sum_{i=0}^{n} v_k$. Nessa nova formulação, queremos responder a seguinte pergunta: Qual é o menor peso necessário para se obter uma soma total de valores *V* com os *k* primeiros itens?

Algorithm 1 Algoritmo de Programação Dinâmica para o Problema da Mochila

$V \leftarrow \sum_{k=1}^{n} v_k$
Crie matriz $DP[0 \dots n][0 \dots V]$
for $x = 1$ **até** V **do**
    $DP[0][x] \leftarrow \infty$
end for
$DP[0][0] \leftarrow 0$
for $k = 1$ **até** n **do**
    for $x = 1$ **até** V **do**
        if $v_k > x$ **then**
            $DP[k][x] \leftarrow DP[k-1][x]$
        else
            $DP[k][x] \leftarrow \min\left(DP[k-1][x],\ w_k + DP[k-1][x - v_k]\right)$
        end if
    end for
end for
return $\max\{x \mid DP[n][x] \leq W\}$

Como vemos, o custo do algoritmo é dominado pelo preenchimento da matriz. Portanto, o custo do algoritmo é $O(nV) = O(n^2 v_{max})$. Note que essa formulação é útil quando a soma dos valores é pequena. Podemos tirar vantagem disso para o nosso algoritmo aproximativo, alterando a escala de valores grandes. Em outras palavras, usamos uma constante μ para converter os valores $v'_i = \lfloor \frac{v_i}{\mu} \rfloor$. Depois, resolvemos o problema utilizando o mesmo algoritmo. Por fim, a solução aproximada é obtida multiplicando-se por μ o valor máximo X encontrado pelo algoritmo de DP cuja restrição de peso é satisfeita (i.e., $DP[n][X] \leq W$).

2.2.1. Fator de aproximação

Entretanto como queremos que o algoritmo tenha um fator de aproximação ε, μ precisa ser uma função f(ε). Já que v_{max} pode estar ou não na solução, então $V^* \geq v_{max}$. Isso implica que, $n\mu = \varepsilon v_{max} \Rightarrow \mu = \frac{\varepsilon v_{max}}{n}$.

Proof. Seja $\mu = \frac{\varepsilon v_{\max}}{n}$ e $v'_i = \left\lfloor \frac{v_i}{\mu} \right\rfloor$. Então:

$$\mu v'_i \leq v_i < \mu(v'_i + 1) \Rightarrow v_i - \mu < \mu v'_i \leq v_i.$$

Seja S a solução retornada pelo algoritmo e O a solução ótima. Como S é ótima para os valores v'_i:

$$\sum_{i \in S} v'_i \geq \sum_{i \in O} v'_i.$$

Logo:

$$\sum_{i \in S} v_i \geq \mu \sum_{i \in S} v'_i \geq \mu \sum_{i \in O} v'_i.$$

Usando $v'_i \geq \frac{v_i - \mu}{\mu}$, temos:

$$\mu \sum_{i \in O} v'_i \geq \sum_{i \in O} (v_i - \mu) = \sum_{i \in O} v_i - n\mu.$$

Substituindo $\mu = \frac{\varepsilon v_{\max}}{n}$:

$$\sum_{i \in S} v_i \geq \sum_{i \in O} v_i - \varepsilon v_{\max}.$$

Como $V^* = \sum_{i \in O} v_i \geq v_{\max}$, concluímos:

$$\sum_{i \in S} v_i \geq V^* - \varepsilon V^* = (1 - \varepsilon) V^*.$$

Portanto, o algoritmo possui fator de aproximação $(1 - \varepsilon)$. □

2.2.2. Complexidade

Proof. O algoritmo utilizado tem complexidade original $O(nV)$, onde $V = \sum_{i=1}^{n} v_i$.

Como aplicamos a transformação dos valores com $v'_i = \left\lfloor \frac{v_i}{\mu} \right\rfloor$ e $\mu = \frac{\varepsilon v_{\max}}{n}$, obtemos:

$$V' = \sum_{i=1}^{n} v'_i = \sum_{i=1}^{n} \left\lfloor \frac{v_i}{\mu} \right\rfloor \leq \sum_{i=1}^{n} \frac{v_i}{\mu} \leq \frac{n v_{\max}}{\mu}.$$

Substituindo $\mu = \frac{\varepsilon v_{\max}}{n}$:

$$V' \leq \frac{n v_{\max}}{\varepsilon v_{\max}/n} = \frac{n^2}{\varepsilon}.$$

Assim, a nova complexidade do algoritmo torna-se:

$$O(nV') = O\left(n \cdot \frac{n^2}{\varepsilon}\right) = O\left(\frac{n^3}{\varepsilon}\right).$$

Portanto, o algoritmo é polinomial tanto em n quanto em $1/\varepsilon$, caracterizando um FPTAS. □

2.3. Algoritmo Guloso Aprimorado 2-aproximativo

Como forma de incrementar nossa análise, decidimos testar outro algoritmo 2-aproximativo a partir da abordagem gulosa. O algoritmo mais guloso ingênuo seria consumir a entrada adicionando itens sempre que possível, no entanto isso pode levar a uma solução longe do ótimo. Aprimorando essa ideia, propomos o seguinte pré-processamento: ordenar a entrada pela razão $\frac{v_i}{w_i}$ do maior para o menor. No entanto, ainda continua tendo falhas e sendo arbitrariamente pior metade da solução ótima. Exemplo: seja a instância $I = \{(2, 1), (100, 100)\}$ e $C = 100$, o algoritmo escolheria o item $i_1 = (2, 1)$ por ter uma razão $\frac{2}{1} > \frac{100}{100}$ e como o peso $1 + 100 > C = 100$ deixaria o item $i_2 = (100, 100)$ de fora, retornando $S = 2$. Mas a solução ótima é item $i_2 = (100, 100)$ retornando $O = 100$, ou seja, a solução $S = O \times \frac{2}{100} = O \times \frac{1}{50}$. Com o intuito de melhorar esse resultado, propomos a seguinte modificação no algoritmo retornar o maior entre a soma gulosa e o item de maior valor que respeite o limite.

2.3.1. Complexidade

O algoritmo gera a lista de densidade em $O(n)$, executa uma ordenação $O(n \log n)$, consome essa entrada $O(n)$ e por fim busca o valor máximo em $O(n)$. Por tanto o custo desse algoritmo é limitado pela ordenação: $O(n \log n)$.

2.3.2. Fator de aproximação

Seja C a capacidade da mochila, S_a o conjunto de itens selecionados pela abordagem gulosa, e $A = \max\left(\sum_{i \in S_a} v_i, v_{\max}\right)$ o valor retornado pelo algoritmo. Seja S_o a solução ótima, com valor $O = \sum_{i \in S_o} v_i$. Então, $A \geq \frac{O}{2}$.

Proof. Ordenamos os itens por densidade não crescente v_i/w_i, e inserimos os itens na mochila até ultrapassar a capacidade C. Seja i_k o primeiro item que não pode ser totalmente adicionado à mochila — este é o **item crítico**.

Note que a soma dos valores da solução gulosa inteira até i_{k-1} (denotada por S_a) mais o valor do item crítico v_k constitui um limite superior para a solução do problema da mochila *fracionária*. Ou seja,

$$\sum_{i \in S_a} v_i + v_k \geq O,$$

pois a solução fracionária é um limite superior para o problema da mochila binária.

Como $v_k \leq v_{\max}$, temos:

$$\sum_{i \in S_a} v_i + v_{\max} \geq \sum_{i \in S_a} v_i + v_k \geq O.$$

O algoritmo retorna:

$$A = \max\left(\sum_{i \in S_a} v_i, v_{\max}\right).$$

Então, como a soma de dois inteiros não-negativos é no máximo o dobro do maior deles, temos:

$$2A \geq \sum_{i \in S_a} v_i + v_{\max} \geq O.$$

Portanto,

$$A \geq \frac{O}{2}.$$

Isso mostra que, no pior caso, o algoritmo retorna uma solução com valor pelo menos metade do ótimo, ou seja, é um algoritmo 2-aproximativo. □

3. Experimentos e Discussão dos Resultados

Esta seção apresentará os resultados obtidos com a execução dos algoritmos em diferentes instâncias do Problema da Mochila. Para os resultados que ainda não foram obtidos, serão deixadas lacunas para preenchimento posterior.

3.1. Configuração Experimental

Os experimentos foram realizados utilizando instâncias do Problema da Mochila 0/1, lidas a partir de arquivos no formato CSV (ou texto similar). O formato esperado dos arquivos é:

- Primeira linha: `[número_de_itens] [capacidade_da_mochila]` (separados por espaço).
- Linhas subsequentes: `[lucro] [peso]` para cada item (separados por espaço).

O algoritmo foi configurado com um limite de tempo de 30 minutos (1800 segundos) por instância.

3.2. Resultados Obtidos (Branch and Bound)

O resultado ótimo foi encontrado em todos os casos, com uma informação relevante que nos casos de teste para os ultimos três arquivos (knapPI_3_2000_1000_1, knapPI_3_5000_1000_1, knapPI_3_10000_1000_1) o tempo de solução explode. Para todos os outros o problema encontra a solução ótima em menos de 2 mintos, para esses o tempo excede os 30 minutos que foram utilizados de limite para a execução de programa (pra fins de brevidade).

No entanto, foi averiguado que, se eles acabam de rodar, o ele atinge o resultado esperado, no entanto o tempo de execução não pode ser salvo nessa execução devido a um erro no log do programa e, devido ao excessivo tempo de resolução, a execução não pode ser repetida.

3.2.1. Limites da Implementação e Relação Tamanho da Instância vs. Desempenho

O algoritmo Branch and Bound, embora garantindo a otimalidade, tem uma complexidade de tempo que pode ser exponencial no pior caso.

- **Instâncias Pequenas a Médias (e.g., 500 itens):** Para instâncias como `knapPI_2_500_1000_1`, o algoritmo encontrou a solução ótima em um tempo extremamente curto (0.0075 segundos). Isso demonstra a eficácia da poda e da heurística gulosa inicial para problemas com um espaço de busca gerenciável.
- **Instâncias Grandes (e.g., 10.000 itens):** Para `knapPI_3_10000_1000_1`, apesar das otimizações (DFS iterativo, heurística gulosa, poda agressiva), o tempo de execução é significativamente maior.

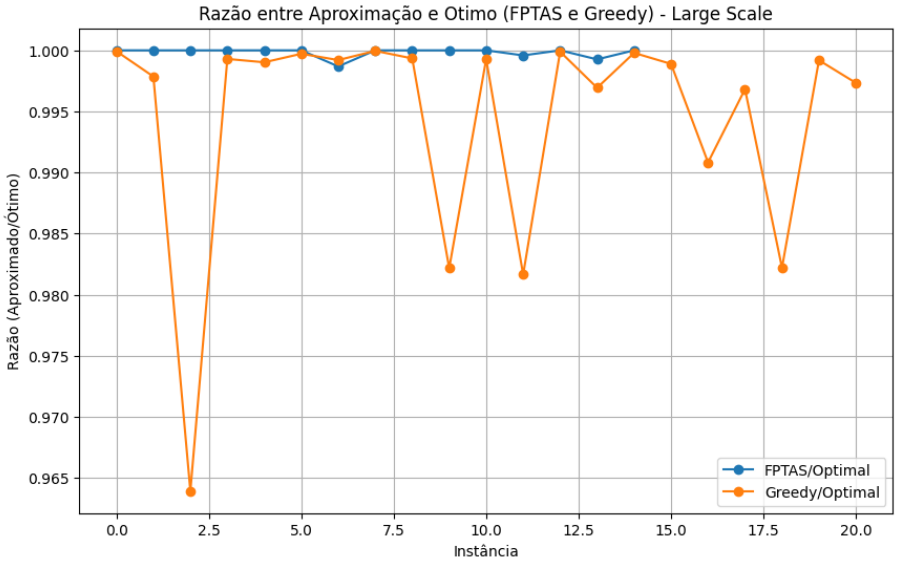
    - **O algoritmo conseguiu rodar em tempo hábil para 2 dos 3 casos com mais de 10k itens, mas para o ultimo, não foi capaz. isso indica que ainda que exista o problema da complexidade é preciso também considerar se a configuração do programa é de melhor caso pra heurística utilizada ou pior caso**
    - **A complexidade apenas piora com o maior número de instancias, no entanto, é possível observar que a configuração das instancias tem um efeito considerável, de modo que diferentes heurísticas podem resultar em tempos consideravelmente diferentes de execução, ainda que a complexidade seja a mesma**

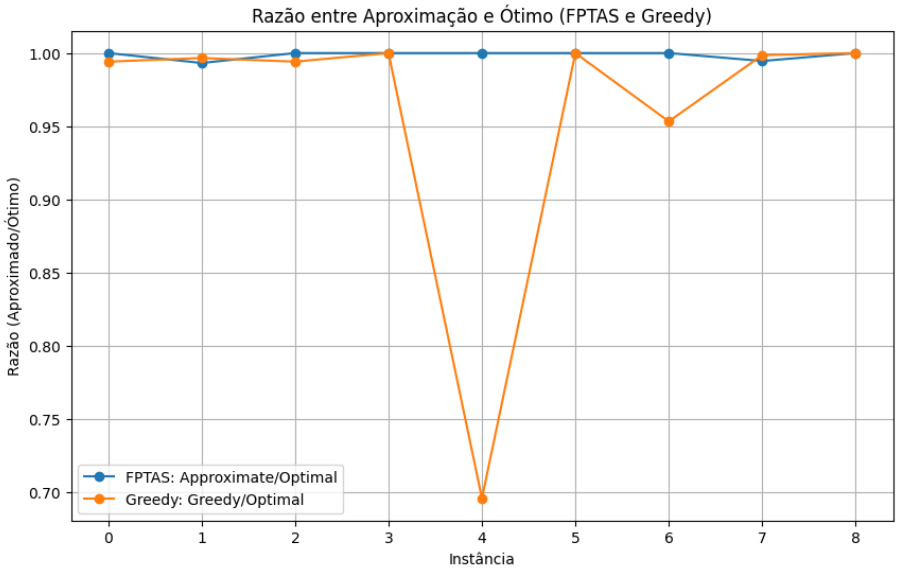
3.2.2. Erro Produzido pelo Algoritmo

Para o algoritmo Branch and Bound, o "erro produzido" é **zero**, pois ele é um algoritmo exato que garante a obtenção da solução ótima global do problema. Quaisquer desvios da solução ótima ocorreriam apenas se o algoritmo fosse interrompido prematuramente (e.g., por um limite de tempo), caso em que ele retornaria a melhor solução *encontrada até aquele ponto*.

3.3. Resultados Obtidos (Aproximativo)

O algoritmo FPTAS não teve um bom desempenho. A pesar de escolher um fator de aproximação $\varepsilon = 0,5$ para garantir um fator de aproximação 2, para as instancias que o FPTAS conseguiu rodar foi muito próximo do ótimo. No entando o tempo de execução desse algoritmo se provou pior do que o branch-and-bound para casos de larga escala. Nos piores casos knapPI_1_5000_1000_1, ou com entrada maior, o algoritmo não executou por falta de memoria, isso se deve a natureza da programação dinâmica em criar uma matriz n. Enquanto o algoritmo greedy conseguiu rodar para todas as entradas. Esse algoritmo foi muito eficiente e, na maioria dos casos, satisfatório na saída. Onde, em poucas instâncias o resultado foi inferior a 95% do ótimo. Comparando as duas imagens podemos ver que o FPTAS foi mais próximo do ótimo do que o guloso.





4. Discussão Final

- É possível observar que o algoritmo ”greedy” é constantemente mais curto que o algoritmo ”branch and bound” e frequentemente encontra o resultado ótimo, assim como é possível observar na tabela2 e na tabela 1. o que indica que. para valores baixos é uma boa aproximação
- Em relação aos valores do FPTAS, ainda que ele encontrem os valores de aproximação próximos ou igual ao ótimo, ele demora muito mais. Dessa forma observamos a importância das heurísticas na implementação do programa guloso.

  booktabs siunitx

References

Ortega, Johny. *Instances for the 0/1 Knapsack Problem*. Universidad del Cauca, Grupo de Inteligencia Computacional (GIC), 2012. Disponível em: `http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/`. Acesso em: 7 jul. 2025.

Table 1. Tabela de resultado por algoritmo

Instância	Optimal	B&B	FPTAS	Greedy	Escala
knapPI_1_10000_1000_1	563 647	563 647	NaN	563 605	Large Scale
knapPI_1_1000_1000_1	54 503	54 503	54 503	54 386	Large Scale
knapPI_1_100_1000_1	9147	9147	9147	8817	Large Scale
knapPI_1_2000_1000_1	110 625	110 625	110 625	110 547	Large Scale
knapPI_1_200_1000_1	11 238	11 238	11 238	11 227	Large Scale
knapPI_1_5000_1000_1	276 457	276 457	NaN	276 379	Large Scale
knapPI_1_500_1000_1	28 857	28 857	28 857	28 834	Large Scale
knapPI_2_10000_1000_1	90 204	90 204	NaN	90 200	Large Scale
knapPI_2_1000_1000_1	9052	9052	9052	9046	Large Scale
knapPI_2_100_1000_1	1514	1514	1512	1487	Large Scale
knapPI_2_2000_1000_1	18 051	18 051	18 051	18 038	Large Scale
knapPI_2_200_1000_1	1634	1634	1634	1604	Large Scale
knapPI_2_5000_1000_1	44 356	44 356	NaN	44 351	Large Scale
knapPI_2_500_1000_1	4566	4566	4566	4552	Large Scale
knapPI_3_10000_1000_1	146 919	146 919	NaN	146 888	Large Scale
knapPI_3_1000_1000_1	14 390	14 390	14 390	14 374	Large Scale
knapPI_3_100_1000_1	2397	2397	2396	2375	Large Scale
knapPI_3_2000_1000_1	28 919	28 919	28 919	28 827	Large Scale
knapPI_3_200_1000_1	2697	2697	2695	2649	Large Scale
knapPI_3_5000_1000_1	72 505	72 505	NaN	72 446	Large Scale
knapPI_3_500_1000_1	7117	7117	7117	7098	Large Scale
f10_l-d_kp_20_879	1025	1025	1025	1019	Small Scale
f1_l-d_kp_10_269	295	295	293	294	Small Scale
f2_l-d_kp_20_878	1024	1024	1024	1018	Small Scale
f3_l-d_kp_4_20	35	35	35	35	Small Scale
f4_l-d_kp_4_11	23	23	23	16	Small Scale
f6_l-d_kp_10_60	52	52	52	52	Small Scale
f7_l-d_kp_7_50	107	107	107	102	Small Scale
f8_l-d_kp_23_10000	9767	9767	9714	9753	Small Scale
f9_l-d_kp_5_80	130	130	130	130	Small Scale

Table 2. Tabela de tempos de execução (em segundos) por algoritmo

Instância	Branch and Bound	FPTAS	Greedy	Escala
knapPI_1_10000_1000_1	0.969 211	NaN	0.004 046	Large Scale
knapPI_1_1000_1000_1	0.031 999	118.163 981	0.000 374	Large Scale
knapPI_1_100_1000_1	0.006 453	0.107 516	0.000 035	Large Scale
knapPI_1_2000_1000_1	0.149 657	705.586 010	0.000 565	Large Scale
knapPI_1_200_1000_1	0.002 780	0.949 324	0.000 064	Large Scale
knapPI_1_5000_1000_1	0.521 508	NaN	0.001 749	Large Scale
knapPI_1_500_1000_1	0.006 039	13.968 797	0.000 142	Large Scale
knapPI_2_10000_1000_1	6.174 612	NaN	0.003 611	Large Scale
knapPI_2_1000_1000_1	0.034 709	108.011 117	0.000 262	Large Scale
knapPI_2_100_1000_1	0.004 246	0.104 899	0.000 059	Large Scale
knapPI_2_2000_1000_1	0.187 123	880.104 872	0.000 555	Large Scale
knapPI_2_200_1000_1	0.008 752	0.859 728	0.000 053	Large Scale
knapPI_2_5000_1000_1	0.509 068	NaN	0.002 091	Large Scale
knapPI_2_500_1000_1	0.010 090	13.734 642	0.000 124	Large Scale
knapPI_3_10000_1000_1	>120	NaN	0.004 023	Large Scale
knapPI_3_1000_1000_1	44.863 878	130.539 020	0.000 266	Large Scale
knapPI_3_100_1000_1	0.002 299	0.122 578	0.000 033	Large Scale
knapPI_3_2000_1000_1	>120	1135.635 412	0.000 569	Large Scale
knapPI_3_200_1000_1	0.276 635	1.006 117	0.000 046	Large Scale
knapPI_3_5000_1000_1	>120	NaN	0.001 763	Large Scale
knapPI_3_500_1000_1	0.589 515	16.058 624	0.000 126	Large Scale
f10_l-d_kp_20_879	0.000 113	0.001 193	0.000 008	Small Scale
f1_l-d_kp_10_269	0.000 051	0.000 516	0.000 008	Small Scale
f2_l-d_kp_20_878	0.000 118	0.001 046	0.000 007	Small Scale
f3_l-d_kp_4_20	0.000 049	0.000 212	0.000 004	Small Scale
f4_l-d_kp_4_11	0.000 135	0.000 221	0.000 004	Small Scale
f6_l-d_kp_10_60	0.000 224	0.000 294	0.000 004	Small Scale
f7_l-d_kp_7_50	0.000 041	0.000 219	0.000 004	Small Scale
f8_l-d_kp_23_10000	3.129 237	0.001 755	0.000 007	Small Scale
f9_l-d_kp_5_80	0.000 052	0.000 210	0.000 003	Small Scale