RELATÓRIO - TP 2
ALGORITMOS II, UFMG

Ester Sara Assis
2021031785

Júlia Paes de Viterbo
2021032137

ABSTRACT

O estudo de algoritmos na graduação começa desde os primeiros semestres no curso de Ciência da Computação. À medida que são introduzidas soluções cada vez mais complexas, constrói-se a compreensão de que existem problemas para os quais não existe resposta exata sem que se perca a computabilidade da resposta. Nesse contexto, insere-se o estudo de algoritmos que se utilizam de estratégias de corte, como *Backtracking* e *Branch and Bound*, e de metodologias aproximativas, baseadas em heurísticas. Uma das abordagens mais singulares dentre as dos algoritmos aproximativos é uma das possíveis para o Problema da Mochila (Knapsack), em que deseja-se obter uma solução que seja no máximo $X\%$ pior que a ótima, e X torna-se um parâmetro de entrada, além de outras em que o fator de aproximação é constante, a depender apenas das escolhas feitas no processo algorítmico. Nesse sentido, o presente estudo aprofunda-se na investigação desse problema, centrando sua análise tanto no comportamento, quanto no desempenho de diferentes algoritmos para ele, incluindo aquele cuja proximidade da saída com o ótimo depende do parâmetro de entrada.

1. INTRODUÇÃO

Este trabalho investiga o desempenho dos seguintes algoritmos para o Knapsack Problem:

1. Branch and Bound (3);
2. Aproximativo com Parâmetro Variável (4);
3. 2-Aproximativo (5).

e usadas instâncias das fontes: `Instances of 0/1 Knapsack Problem` (*low dimensional*) [1] e `Dataset Large Scale` (*large scale*) [2] como teste.

2. KNAPSACK PROBLEM

2.1 Apresentação

De forma direta, o Problema da Mochila pode ser interpretado como: "dados `n` itens, cada um com um peso e valor associado, e uma mochila com capacidade `W` quilos, deseja-se maximizar o lucro total, respeitando a capacidade da mochila." [3]

O problema, como visto em sala, é NP-Completo, um forte indício de que não há forma polinomial de resolvê-lo em uma máquina de Turing determinística. Dessa forma, as três abordagens de algoritmos implementadas foram testadas em termos de seu desempenho em gasto tempo, uso de memória e qualidade dos resultados.

2.2 Implementação

2.2.1 Construção das Estruturas de Dados

A classe principal, `KnapsackProblemRunner`, é responsável por orquestrar todo o processo de execução e avaliação dos algoritmos para o Problema da Mochila. Sua função é gerenciar a leitura das instâncias, a execução assíncrona dos algoritmos para cada arquivo, o cálculo das métricas de desempenho e, por fim, a geração dos arquivos de saída com os resultados consolidados.

Optou-se pela utilização do modelo *async* em detrimento de técnicas convencionais de *threading* ou *multiprocessing*, uma vez que as soluções concorrentes baseadas em processos apresentaram dificuldades no compartilhamento de estado entre o processo pai e os processos filhos, exigindo mecanismos adicionais de comunicação, como filas, para sincronizar os atributos do algoritmo, o que aumentava a complexidade e a suscetibilidade a falhas. O uso de *threads*, por sua vez, não possibilitava a interrupção real de algoritmos pesados, apenas a sinalização de tempo limite, mantendo as *threads* ativas indefinidamente. Dessa forma, a adoção do paradigma *async* permitiu a execução concorrente no mesmo contexto de memória, garantindo o monitoramento de tempo limite de forma cooperativa com *asyncio* e evitando a necessidade de mecanismos explícitos de comunicação interprocesso, preservando a consistência dos dados no objeto *Python*. A inclusão dessa abordagem no projeto foi apresentada e aprovada pelo professor da disciplina.

Em nível mais granular, a classe `KnapsackProblem` realiza a leitura e o processamento do arquivo de instância para extrair as informações fundamentais:

1. **Capacidade da Mochila (W)**: O peso máximo que a mochila suporta.
2. **Itens**: Uma lista de itens, onde cada um é representado por seu peso e valor associados.

Esses dados, `n` (número de itens), `W` (capacidade) e a lista de `items`, são armazenados na instância da classe e servem como base para a execução de todos os algoritmos implementados. Em seguida, os itens são lidos, e para cada um, a razão valor/peso é calculada. Uma etapa crucial de pré-processamento é a ordenação da lista de itens

em ordem decrescente com base nessa razão. Essa ordenação é fundamental para a heurística do algoritmo Branch and Bound e para o funcionamento do algoritmo de 2-aproximação.

2.2.2 Execução dos Algoritmos

Para otimizar o tempo de processamento de múltiplas instâncias, foi utilizada a abordagem assíncrona explicada, que é gerenciada pela classe `KnapsackProblemRunner`. Para cada instância, os três algoritmos são instanciados e executados de forma concorrente.

A concorrência é alcançada utilizando a biblioteca `asyncio`. Uma função interna,`execute_with_timeout`, encapsula a chamada de cada algoritmo, impondo um tempo limite de 1800 segundos (30 minutos) através de `asyncio.wait_for`, como especificado no texto de especificação do trabalho. Se um algoritmo excede esse tempo, sua execução é interrompida e ele é marcado como *timeout*. Para garantir que os algoritmos de longa duração não bloqueiem o loop de eventos, o comando `await asyncio.sleep(0)` é utilizado em seus laços internos para ceder o controle de execução periodicamente.

3. BRANCH AND BOUND

3.1 Apresentação

O algoritmo de Branch and Bound para o problema de Knapsack se dá por meio da construção de uma árvore cujos nós originam dois ramos de busca sempre: um que inclui, outro que exclui um item.

Seguindo essa lógica, as podas acontecem por meio da avaliação de lucro máximo, *upper bound*, a ser obtido a partir de cada nó. Já a exploração dos nós se dá pela lógica de **Best-First Search**, na qual uma fila de prioridade é usada para sempre expandir o nó mais promissor da fronteira de busca. Assim, o *bound* decide *se* um nó entra na fila, e a fila de prioridade decide *qual* nó sai da fila primeiro. Além disso, ramos cuja soma dos pesos dos itens ultrapassem o peso máximo da mochila sempre são podados. A **Figura 1** mostra um exemplo de execução do algoritmo em um conjunto de entrada simples (**Tabela 1**).

Table 1. Exemplo de uma instância do Problema da Mochila com capacidade $W = 10$.

Item	Peso	Valor	Valor/Peso
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

3.2 Implementação

A implementação do algoritmo foi feita com base no pseudocódigo abaixo (**Algoritmo 1**), adaptado do slide da aula 12 [3].

Algorithm 1 Branch and Bound Knapsack adaptado

procedure BNB-KNAPSACK($items, W, n$)
  $best\_value \leftarrow 0$
  $Q \leftarrow$ PriorityQueue()
  $root \leftarrow$ Node$(-1, 0, 0)$
  $Q.put(root)$
  while Q not empty **do**
    $node \leftarrow Q.get()$
    if $node.level = n - 1$ **then**
      continue
    end if
    $next\_level \leftarrow node.level + 1$
    $v_{with} \leftarrow node.value + items[next\_level].value$
    $w_{with} \leftarrow node.weight + items[next\_level].weight$
    if $w_{with} \leq W$ **and** $v_{with} > best\_value$ **then**
      $best\_value \leftarrow v_{with}$
    end if
    $with\_node \leftarrow$ Node$(next_level, v_{with}, w_{with})$
    if $bound(with\_node) > best\_value$ **then**
      $Q.put(with\_node)$
    end if
    $without\_node \leftarrow$ Node$(next_level, node.value, node.weight)$
    if $bound(without\_node) > best\_value$ **then**
      $Q.put(without\_node)$
    end if
    await task switch ▷ Para execução assíncrona
  end while
  return $best\_value$
end procedure

Algorithm 2 Cálculo do Limite Superior (Bound)

function BOUND($node, items, W, n$)
  if $node.weight \geq W$ **then**
    return 0
  end if
  $bound \leftarrow node.value$
  $current\_weight \leftarrow node.weight$
  $k \leftarrow node.level + 1$
  while $k < n$ **and** $current\_weight + items[k].weight \leq W$ **do**
    $current\_weight \leftarrow current\_weight + items[k].weight$
    $bound \leftarrow bound + items[k].value$
    $k \leftarrow k + 1$
  end while
  if $k < n$ **then**
    $bound \leftarrow bound + (W - current\_weight) \cdot items[k].ratio$
  end if
  return $bound$
end function

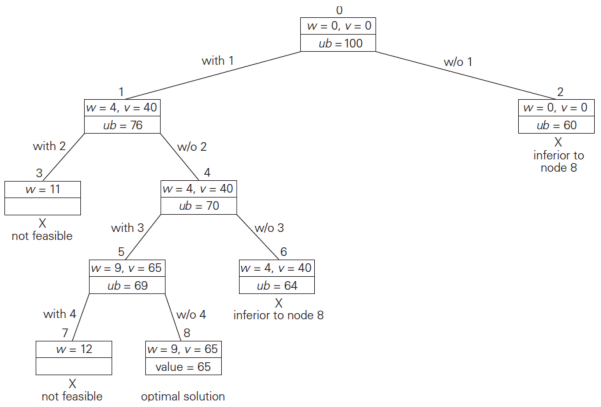

Figure 1. `LEVITIN` [5], cap. 12, sec. 2

A implementação do algoritmo utiliza duas estruturas de dados centrais: a classe `Node`, para encapsular o estado de cada nó (nível, valor e peso), e uma `PriorityQueue` para gerenciar os nós a serem explorados. A classe `Node` foi definida como um *dataclass* ordenável, o que faz com que a fila de prioridade organize os nós com base na tupla de seus atributos (nível, valor e peso).

O procedimento, detalhado no **Algoritmo 1**, itera sobre a fila de prioridade. A cada passo, o nó mais promissor é removido, e seus dois filhos ("com" e "sem" o próximo item) são gerados. Um nó filho só é adicionado à fila se o seu limite superior, calculado pela função `bound` (**Algoritmo 2**), indicar que aquele caminho ainda é promissor.

3.3 Métricas

Para calcular as métricas desse algoritmo, foi utilizado:

- **Tempo**: O tempo de execução é aferido utilizando `time.process_time()` para registrar o tempo de CPU consumido pelo algoritmo.
- **Espaço**: O uso de memória é estimado com base no tamanho máximo que a fila de prioridade atinge durante a execução, multiplicado pelo tamanho em bytes de um único objeto `Node`.
- **Qualidade**: O erro relativo é calculado pela classe controladora, conforme o método padrão definido no projeto.

4. APROXIMATIVO COM PARÂMETRO VARIÁVEL

4.1 Apresentação

Para a primeira solução aproximativa do problema, recorreu-se a uma classe de algoritmos de aproximação conhecida como Esquema de Aproximação Totalmente Polinomial (FPTAS, do inglês Fully Polynomial-Time Approximation Scheme). Esses algoritmos se destacam por permitirem que o usuário defina um parâmetro de qualidade $epsilon$, que garante que a solução encontrada seja no máximo $1epsilon$ vezes o valor da solução ótima.

A estratégia central do FPTAS para a mochila é baseada em uma formulação de programação dinâmica que, embora pseudo-polinomial, é eficiente quando os valores dos itens são pequenos. Para contornar o problema de instâncias com valores grandes, o algoritmo altera a escala desses valores, arredondando-os para baixo com base em um fator de escala derivado de $epsilon$. Após resolver o problema com os valores reduzidos, a solução é convertida de volta para a escala original, resultando em uma aproximação controlada.

A complexidade de tempo final do algoritmo é polinomial tanto no tamanho da entrada quanto em $1/epsilon$, o que implica que, quanto menor for o erro desejado, maior será o tempo de execução.

4.2 Implementação

A implementação do FPTAS se baseia em uma formulação de programação dinâmica que responde à pergunta: "qual é o menor peso necessário para se obter um determinado valor?". O algoritmo, apresentado em resumo no **Algoritmo 3** e detalhado nos slides [4], aplica essa lógica sobre os valores escalados dos itens.

Algorithm 3 FPTAS para o Problema da Mochila

procedure FPTAS-KNAPSACK($items, W, n, \epsilon$)
  $v_{max} \leftarrow \max(\{i.value \mid i \in items\})$
  $\mu \leftarrow (\epsilon \cdot v_{max})/n$
  for all item i em $items$ **do**
    $v'_i \leftarrow \lfloor i.value/\mu \rfloor$
  end for
  $V' \leftarrow \sum_{i=1}^{n} v'_i$
  $DP \leftarrow$ array de tamanho $(n+1) \times (V'+1)$
  for $i = 1$ to V' **do**
    $DP[0][i] \leftarrow \infty$
  end for
  for $k = 1$ to n **do**
    for $X = 1$ to V' **do**
      if $v'_k > X$ **then**
        $DP[k][X] \leftarrow DP[k-1][X]$
      else
        $DP[k][X] \leftarrow \min(DP[k-1][X], items[k].weight + DP[k-1][X - v'_k])$
      end if
    end for
  end for
  $X_{best} \leftarrow \max(\{X \mid DP[n][X] \leq W\})$
  return $X_{best} \cdot \mu$
end procedure

É importante notar que a implementação prática deste trabalho segue a estratégia teórica dos slides com uma otimização de espaço adicional: enquanto a teoria apresentada nos slides descreve uma tabela de programação dinâmica bidimensional (`DP[k][X]`), o código utiliza uma tabela unidimensional (`dp[v]`).

Nessa abordagem otimizada:

- O *array* `dp` armazena diretamente o menor peso

necessário para se atingir um determinado valor escalonado v.

- Para garantir que a lógica da recorrência seja preservada (ou seja, para que um item não seja usado múltiplas vezes na mesma etapa de cálculo), o laço interno que atualiza a tabela `dp` itera de forma decrescente.

Essa técnica produz um resultado idêntico ao da abordagem bidimensional, mas com uma necessidade de memória significativamente menor, que depende apenas do valor de V' e não de $n \times V'$.

Sobre o fator aproximativo, na seção de experimentos 6 será mostrado como foi procedido com a definição de $epsilon$.

4.3 Métricas

Para calcular as métricas desse algoritmo, foi utilizado como base o estudo de sua complexidade e garantia de aproximação teóricas, apresentadas nos slides [4].

- **Tempo**: A complexidade de tempo do algoritmo é dominada pelo preenchimento da tabela de programação dinâmica. Conforme demonstrado na referência de aula, obtém-se a complexidade de tempo final do FPTAS como se segue, seja V a soma dos valores dos itens após o escalonamento, e V', a soma de todos esses novos valores escalonados (espera-se um número muito menor que o V original):

$$T(n, \epsilon) = O(n \cdot V') = O\left(n \cdot \frac{n^2}{\epsilon}\right) = O\left(\frac{n^3}{\epsilon}\right) \tag{1}$$

  Na prática, o tempo de execução da CPU é mensurado de forma precisa utilizando a função `time.process_time()`.

- **Espaço**: A implementação em código adota a otimização explicada que, em vez de uma tabela bidimensional de tamanho O(nV), utiliza um único *array* unidimensional, `dp`, cujo tamanho é diretamente proporcional a V'. Considerando que, no **Algoritmo 3**, $V' = O\left(\frac{n^2}{\epsilon}\right)$, a complexidade de espaço seria também proporcional a isso. No código, isso é *estimado* com base no principal consumidor de espaço do algoritmo multiplicando-se o número de elementos no *array* `dp` por uma constante de 24 bytes, que representa o tamanho aproximado de cada entrada na tabela.

- **Qualidade**: O algoritmo garante um fator de aproximação de $(1 - \epsilon)$. Na prática, o erro relativo é calculado pela classe controladora `KnapsackProblem`. O método lê o valor ótimo de um arquivo de solução correspondente e o compara com o valor retornado pelo algoritmo, utilizando a fórmula $((valor\ timo\ \ valor\ obtido)/\ valor\ timo)$.

5. 2-APROXIMATIVO

5.1 Apresentação

Por fim, o algoritmo de 2-aproximação para o Problema da Mochila escolhido segue uma heurística gulosa cuja estratégia se baseia em uma comparação de duas abordagens distintas para preencher a mochila:

1. Uma solução puramente gulosa, que consiste em adicionar itens à mochila em ordem decrescente de sua densidade de valor (valor/peso), ordem sabida desde a etapa de pré-processamento 2.2.1, até que não seja possível adicionar mais nenhum.

2. Uma solução que considera apenas o item de maior valor singular que, sozinho, caiba na capacidade da mochila.

A garantia de que este algoritmo é 2-aproximativo pode ser demonstrada de forma sucinta.

Seja V_{OPT} o valor da solução ótima. Seja A o valor retornado pelo nosso algoritmo, onde $A = \max(V_g, V_m)$, sendo V_g o valor da solução gulosa e V_m o valor do item de maior valor singular. O objetivo é provar que $V_{OPT} \leq 2A$.

Prova: Consideremos o conjunto de itens da solução ótima. Se o algoritmo guloso encontra essa solução, a prova é trivial. Caso contrário, significa que o algoritmo guloso parou de adicionar itens porque o próximo item, que chamaremos de $k + 1$, não cabia mais na mochila.

Neste ponto, o valor acumulado pela parte gulosa é V_g. Como os itens são processados em ordem decrescente de densidade (valor/peso), qualquer solução ótima está limitada pelo valor da solução gulosa mais o valor do primeiro item descartado. Portanto, temos a seguinte inequação (a desigualdade é estrita porque o item $k + 1$ não cabe por inteiro):

$$V_{OPT} < V_g + v_{k+1} \tag{2}$$

O nosso algoritmo retorna $A = \max(V_g, V_m)$. Pela definição, sabemos que:

1. $A \geq V_g$
2. $A \geq V_m$

O item $k+1$ é um único item cujo peso é, por definição, menor ou igual à capacidade total W. O valor V_m é o valor do item de maior valor que cabe na mochila, portanto, $V_m \geq v_{k+1}$.

Combinando esses fatos, podemos limitar a soma $V_g + V_m$:

$$V_g + V_m \leq A + A = 2A \tag{3}$$

Agora, juntando as inequações (1) e (2):

$$V_{OPT} < V_g + v_{k+1} \leq V_g + V_m \leq 2A \tag{4}$$

Isso nos leva à conclusão de que $V_{OPT} \leq 2A$, ou, de forma equivalente, $A \geq \frac{1}{2} V_{OPT}$. Isso prova que o algoritmo é 2-aproximativo.

Uma prova e análise mais detalhada deste algoritmo podem ser encontradas na Seção 11.8 do livro *Algorithm Design* de Kleinberg e Tardos [6].

5.2 Implementação

Conforme detalhado no Algoritmo 4, a implementação primeiro calcula o valor total obtido pela abordagem gulosa. Em seguida, encontra o item de maior valor único que respeita a capacidade da mochila. Por fim, retorna o máximo entre os dois valores computados.

Algorithm 4 Algoritmo de 2-Aproximação para o Problema da Mochila

procedure TWO-APPROX-KNAPSACK($items, W$)
  $valor\_guloso \leftarrow 0$
  $peso\_atual \leftarrow 0$
  for cada item i em $items$ ▷ Itens pré-ordenados por densidade **do**
    if $peso\_atual + i.weight \leq W$ **then**
      $peso\_atual \leftarrow peso\_atual + i.weight$
      $valor\_guloso \leftarrow valor\_guloso + i.value$
    end if
  end for
  $valor\_max\_singular \leftarrow \max(\{i.value \mid i \in items \text{ e } i.weight \leq W\} \cup \{0\})$
  return $\max(valor\_guloso, valor\_max\_singular)$
end procedure

5.3 Métricas

Para calcular as métricas desse algoritmo, foi utilizado:

- **Tempo**: O tempo de execução da CPU é aferido através da função `time.process_time()` no início e no fim da execução do algoritmo.
- **Espaço**: O consumo de memória é considerado constante. A implementação não aloca estruturas de dados adicionais que dependam do tamanho da entrada, utilizando apenas um número fixo de variáveis de controle.
- **Qualidade**: O erro relativo da solução é calculado de forma padronizada pela classe controladora `KnapsackProblem`, conforme o método geral definido para o projeto.

6. EXPERIMENTOS

As análises entre instâncias de pequena e larga escala (*low dimensional* e *large scale* especificadas pelas fontes do trabalho) foram divididas como indicam as subseções. Cada instância foi rodada uma vez para cada algoritmo por até 30 minutos de execução. Caso o teste excedesse esse tempo, o valor padrão NA deveria substituir o valor esperado de resultado para aquele teste.

Para a análise do algoritmo FPTAS 4, os experimentos foram conduzidos utilizando três valores para o seu parâmetro de precisão, $epsilon$. Cada instância do problema foi resolvida com $epsilon$ configurado para 0.8, 0.5 e 0.1, com o objetivo de observar empiricamente o (*trade-off*) entre o tempo de execução e a qualidade da solução, analisando como a diminuição do erro garantido impacta o desempenho prático do algoritmo.

Na seção de Apêndice 8 está indicado onde encontrar as tabelas com os valores mostrados nos gráficos de forma mais clara, caso os gráficos não sejam suficientes - ao final do trabalho elas também serão adicionadas por completude.

6.1 Análise de Tempo para Instâncias Pequenas

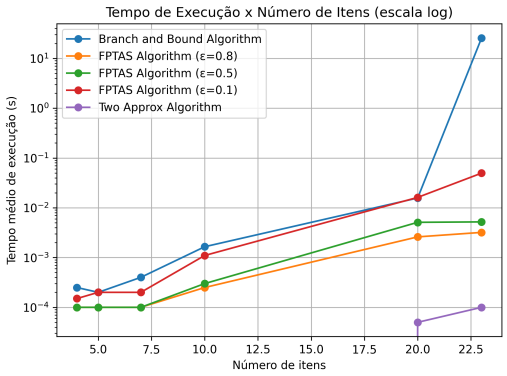

Figure 2. Comparação de Tempo de Execução para Instâncias Pequenas

Pela **Figura 2**, percebe-se que, até as instâncias de tamanho até 20 dentre as disponíveis, à exceção do Algoritmo 2-Aproximativo, cujo tempo de execução foi próxima a 0 segundo, todos os algoritmos tiveram um desempenho parecido, com comportamento de crescimento similar, ainda que, para os FPTAS, quanto melhor o fator de aproximação, maior a espera da execução, como esperado. Contudo, a partir do valor superior a 20 itens, a média de execução do Algotimo Branch and Bound foi no mínimo 10 vezes pior do que todas as demais, ao passo que o desempenho do 2-Aproximativo foi excelente, sendo mais de 10 vezes melhor do que o menor tempo mais próximo ao dele.

6.2 Análise de Memória para Instâncias Pequenas

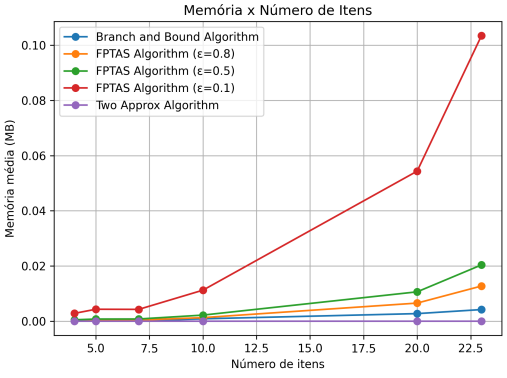

Figure 3. Comparação de Uso de Memória para Instâncias Pequenas

Pela **Figura 3**, percebe-se uma leve vantagem do algoritmo Branch and Bound em relação aos FPTAS em todas as instâncias, mantendo seu uso de memória equilibrado mesmo com aumento delas. Isso é esperado, uma vez que o mecanismo de poda do mesmo economiza bastante memória. É interessante notar especialmente a curva do FPTAS-0.1, que teve tendência ao rápido crescimento, cada vez mais acentuado, logo a partir dos menores tamanhos de instâncias. Mais um vez, o 2-Aproximativo mostrou-se superior aos demais, mas de forma menos evidente do que no caso do tempo de execução.

6.3 Análise de Qualidade para Instâncias Pequenas

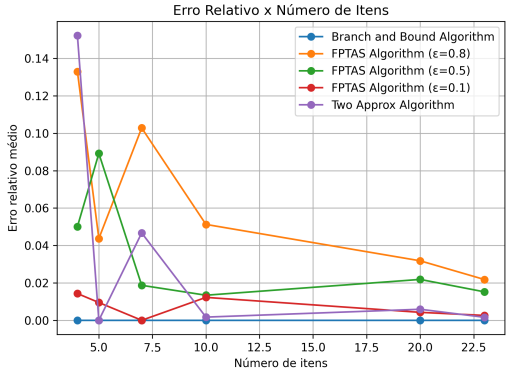

Figure 4. Comparação de Erro Relativo para Instâncias Pequenas

Pela **Figura 4**, percebe-se que para instâncias pequenas, todos os algoritmos chegam bem próximos do valor ótimo, embora os comportamentos das curvas sejam bem diferentes. Notavelmente, para o Branch and Bound, por ser um algoritmo exaustivo, o ótimo sempre será encontrado, então seu erro relativo sempre será 0. No caso dos outros algoritmos, vê-se que o FPTAS-0.1 aqui se sai melhor do que os com fatores aproximativos maiores, o que faz sentido, já que o erro é ainda mais limitado.

6.4 Outras Análises para Instâncias Pequenas

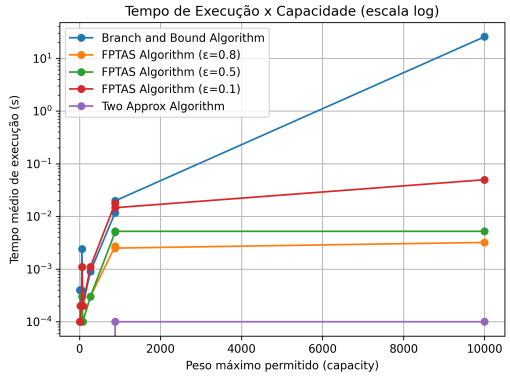

Figure 5. Comparação de Tempo de Execução por W Máximo para Instâncias Pequenas

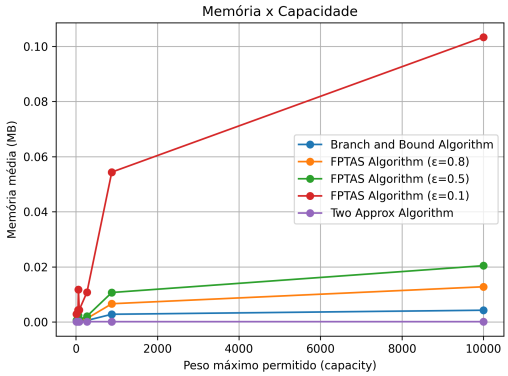

Figure 6. Comparação de Uso de Memória por W Máximo para Instâncias Pequenas

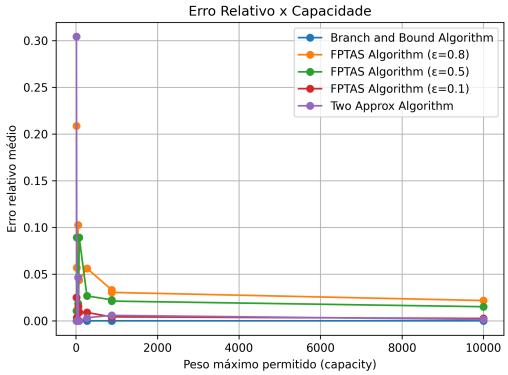

Figure 7. Comparação de Erro Relativo por W Máximo para Instâncias Pequenas

Pela **Figura 5**, percebe-se que, à medida que W aumenta, o comportamento do tempo de execução médio cresce de forma parecida em termos de comparação entre algoritmos com o que se viu em relação ao aumento do tamanho das instâncias.

Na **Figura 6**, a memória também se comporta de forma parecida com o visto na **Figura 3** à medida que o peso máximo aumenta, e, por fim, pela **Figura 7**, já percebe-se que o erro relativo médio de todos os algoritmos (exceto o do Branch and Bound, que é sempre nulo, como explicado) também tem padrões comparáveis aos do gráfico com *Número de itens* no eixo Y ao invés do *Peso máximo*.

6.5 Análise de Tempo para Instâncias Grandes

Pela **Figura 8**, vê-se que, para as instâncias grandes, apenas o 2-Aproximado teve valores médios calculados para todas as instâncias, *i.e.* o tempo máximo de rodagem (30 minutos) foi excedido em algum momento por todos os demais. Além disso, independente do tamanho da amostra, aquele algoritmo teve melhor desempenho do que estes, não levando mais do que alguns segundos para calcular um resultado para qualquer tamanho de teste.

Dentre o Branch and Bound e os FPTAS, apesar do observado, vê-se uma tendência de que o primeiro agora demore menos tempo do que os de parâmetro variável, os quais, novamente, seguem o padrão esperado de que

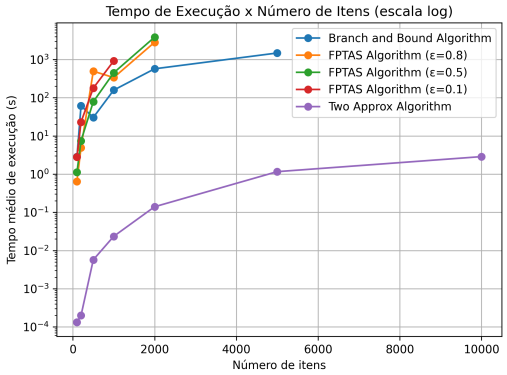

Figure 8. Comparação de Tempo de Execução para Instâncias Grandes

quanto melhor a aproximação, maior também é o tempo de execução à medida que a instância aumenta de tamanho.

6.6 Análise de Memória para Instâncias Grandes

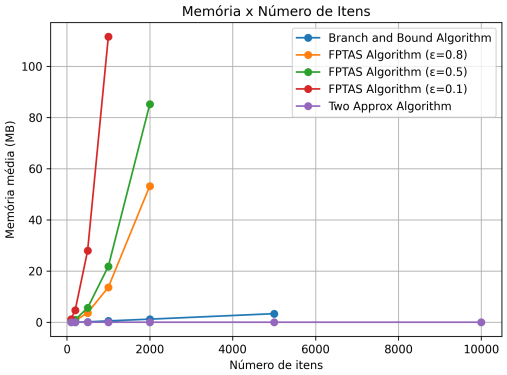

Figure 9. Comparação de Tempo de Execução para Instâncias Grandes

Pela **Figura 9**, percebe-se que, ainda que não haja informações para todos os tamanhos de instâncias para os algoritmos que não o 2-Aproximativo, a tendência de explosão de memória dos FTPAS se mantém, sendo a inclinação mais acentuada para menores $epsilons$. Enquanto isso, uso de memória é mínimo para o 2-Aproximativo.

6.7 Análise de Qualidade para Instâncias Grandes

Pela **Figura 10**, vê-se que, para instâncias grandes, o erro relativo a que qualquer um dos algoritmos chega é relativamente pequeno (pouco mais de 0.020 unidades de valor). Para os casos em que há informações sobre todos os algoritmos, observa-se a tendência de que o erro diminua a uma taxa cada vez menor - o FPTAS-0.1 chega bem próximo ao valor nulo (que é constante para o Branch and Bound, obviamente) com bem menos instâncias do que o faz o 2-Aproximativo, que decresce mais lentamente do que os demais, se aproximando do zero com cerca de 10 vezes mais instâncias do que este. Em comparação aos demais FPTAS, entretanto, o 2-Aproximativo tem desempenho bastante parecido nos pontos em que o tempo não excedeu o limite para os variáveis.

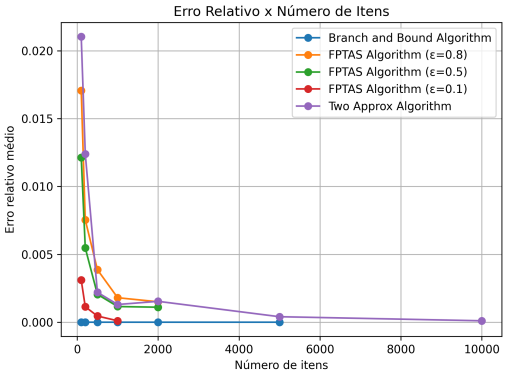

Figure 10. Comparação de Erro Relativo para Instâncias Grandes

6.8 Outras Análises para Instâncias Grandes

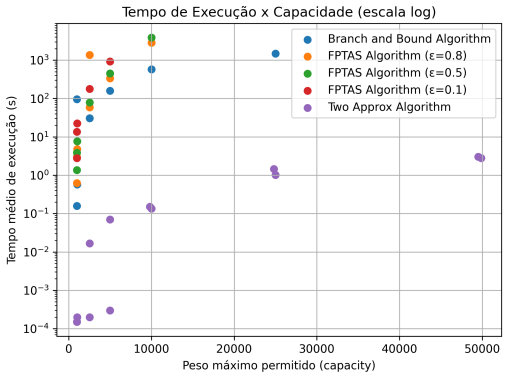

Figure 11. Comparação de Tempo de Execução por W Máximo para Instâncias Grandes

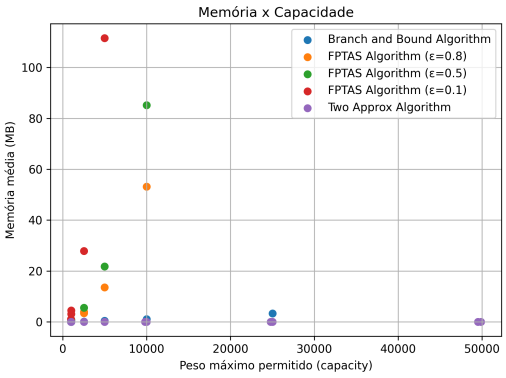

Figure 12. Comparação de Uso de Memória por W Máximo para Instâncias Grandes (Escala Reduzida)

Pela **Figura 11**, observa-se um semelhança grande entre o comportamento dos pontos de média de tempo de execução entre este e o gráfico da **Figura 8**. Da mesma forma,

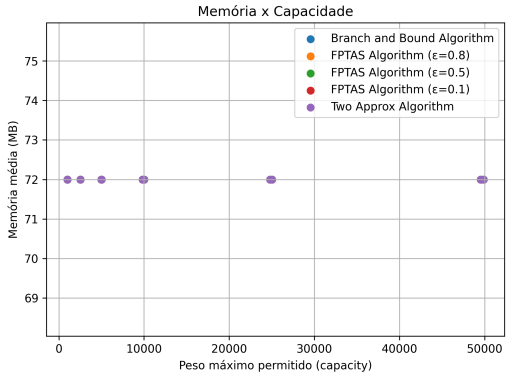

Figure 13. Comparação de Uso de Memória por W Máximo para Instâncias Grandes (Escala Ampliada)

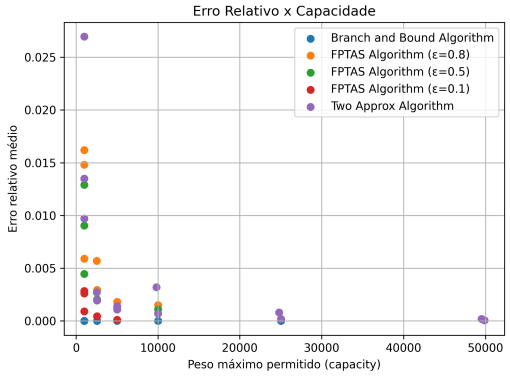

Figure 14. Comparação de Erro Relativo por W Máximo para Instâncias Grandes

pela **Figura 12** e **Figura 13**, o comportamento do uso de memória à medida que o peso máximo cresce entre os algoritmos também é similar ao observado à medida que o tamanho das instâncias cresce para os exemplos testados. Por fim, pela **Figura 14**, observa-se também semelhança com o caso em que, ao invés de `W`, analisa-se o crescimento do tamanho dos casos de teste.

7. CONCLUSÃO

Com base em todos os gráficos expostos, é possível concluir que a melhor escolha algorítmica para o Knapsack Problem tem intrínseca relação com o tamanho da instância trabalhada. Para problemas com baixa dimensionalidade, o erro relativo nulo da abordagem por Branch and Bound paga pela rodagem de alguns segundos do algoritmo. Contudo, com alguns itens a mais em jogo, o custo em tempo já fica desfavorável, e pagar o preço da aproximação é possivelmente um caminho alternativo mais viável.

Pensando então nos demais algoritmos, aqui o gargalo continua a mudar com base em recursos de tempo e memória disponíveis. A partir de pouco mais de mil itens na lista, qualquer algoritmo de FPTAS leva cerca de meia hora ou além isso para calcular um resultado. Contudo, a qualidade média da aproximação dos com parâmetros de 0.5 e 0.8 não se mostra tão mais vantajosa do que a abordagem proposta pelo Algortimo 2-Aproximativo, que é bem superior em termos de tempo de execução e uso de memória. A grande diferença está nas instâncias ainda grandes, mas menores que mil instâncias, em que, embora o erro relativo médio do FPTAS-0.1 seja muito próximo do ideal, o uso de memória nessa abordagem demanda significativamente mais disponibilidade física do computador do que usando a lógica 2-aproximativa. Ainda, o tempo de espera é até 10000 vezes menor usando esta abordagem menos calibrada.

Tendo esse conhecimento em mente, o presente trabalho foi uma ferramenta positiva de estudo e aplicação do conteúdo visto em sala. Apesar de ter levado bastante tempo para que todas as instâncias fossem testadas, o uso de técnicas assíncronas possibilitou relativo aceleramento do processo, e a produção do relatório custou menos na parte de análises do que na de explicação e formalização das intenções pretendidas. Assim, conclui-se o projeto a que se propôs o grupo, considerando concluídos os objetivos com que ele se comprometeu ao iniciá-lo.

8. APÊNDICE

As implementações, tabelas e gráficos deste trabalho estão disponíveis no repositório GitHub: https://github.com/estersassis/knapsack-algorithms.

8.1 Instruções de Execução

O `README.md` do projeto contém as indicações de instruções necessárias para a execução do programa na seção `Instruções de Execução`.

9. REFERENCES

[1] ORTEGA, Johny. Universidad del Cauca. Instâncias 0-1 Knapsack Problem (Low-Dimensional). Disponível em: `http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/`. Acesso em: 06 jul. 2025.

[2] sc0v1n0. Kaggle. Large-Scale 0-1 Knapsack Problems. Disponível em: `https://www.kaggle.com/datasets/sc0v1n0/large-scale-01-knapsack-problems`. Acesso em: 06 jul. 2025.

[3] VIMIEIRO, Renato. DCC/ICEx/UFMG. Aula 12 – Soluções exatas para problemas difíceis (branch-and-bound). DCC207 – Algoritmos 2. Disponível em: `https://virtual.ufmg.br/20242/pluginfile.php/331282/mod_folder/content/0/aula12-branch-and-bound.pdf?forcedownload=1`. Acesso em: 06 jul. 2025

[4] VIMIEIRO, Renato. DCC/ICEx/UFMG. Aula 15 – Soluções aproximadas para problemas difíceis (Parte 3). DCC207 – Algoritmos 2. Disponível em: `https://virtual.ufmg.br/20251/pluginfile.`

Table 2. Instâncias de Baixa Dimensão: Erro relativo médio por instância

Instância	Branch and Bound	Two Approx Algorithm	FPTAS $\varepsilon = 0.8$	FPTAS $\varepsilon = 0.5$	FPTAS $\varepsilon = 0.1$
f10_l-d_kp_20_879	0.0000	0.0059	0.0305	0.0212	0.0043
f1_l-d_kp_10_269	0.0000	0.0034	0.0563	0.0268	0.0091
f2_l-d_kp_20_878	0.0000	0.0059	0.0331	0.0225	0.0042
f3_l-d_kp_4_20	0.0000	0.0000	0.0571	0.0893	0.0036
f4_l-d_kp_4_11	0.0000	0.3043	0.2087	0.0109	0.0250
f6_l-d_kp_10_60	0.0000	0.0000	0.0462	0.0000	0.0154
f7_l-d_kp_7_50	0.0000	0.0467	0.1028	0.0187	0.0000
f8_l-d_kp_23_10000	0.0000	0.0016	0.0218	0.0152	0.0026
f9_l-d_kp_5_80	0.0000	0.0000	0.0437	0.0892	0.0095

Table 3. Instâncias de Baixa Dimensão: Consumo de memória médio por instância (em bytes)

Instância	Branch and Bound	Two Approx Algorithm	FPTAS $\varepsilon = 0.8$	FPTAS $\varepsilon = 0.5$	FPTAS $\varepsilon = 0.1$
f10_l-d_kp_20_879	2,928	72	6,936	11,208	57,024
f1_l-d_kp_10_269	576	72	1,320	2,184	11,280
f2_l-d_kp_20_878	2,928	72	6,912	11,184	56,976
f3_l-d_kp_4_20	144	72	384	576	3,072
f4_l-d_kp_4_11	192	72	336	600	2,976
f6_l-d_kp_10_60	1,344	72	1,488	2,544	12,384
f7_l-d_kp_7_50	192	72	480	888	4,536
f8_l-d_kp_23_10000	4,464	72	13,392	21,408	108,384
f9_l-d_kp_5_80	144	72	576	864	4,584

Table 4. Instâncias de Baixa Dimensão: Tempo de execução médio por instância (em segundos)

Instância	Branch and Bound	Two Approx Algorithm	FPTAS $\varepsilon = 0.8$	FPTAS $\varepsilon = 0.5$	FPTAS $\varepsilon = 0.1$
f10_l-d_kp_20_879	0.0197	0.0001	0.0025	0.0052	0.0146
f1_l-d_kp_10_269	0.0009	0.0000	0.0003	0.0003	0.0011
f2_l-d_kp_20_878	0.0117	0.0000	0.0027	0.0050	0.0177
f3_l-d_kp_4_20	0.0001	0.0000	0.0001	0.0001	0.0002
f4_l-d_kp_4_11	0.0004	0.0000	0.0001	0.0001	0.0001
f6_l-d_kp_10_60	0.0024	0.0000	0.0002	0.0003	0.0011
f7_l-d_kp_7_50	0.0004	0.0000	0.0001	0.0001	0.0002
f8_l-d_kp_23_10000	25.6479	0.0001	0.0032	0.0052	0.0496
f9_l-d_kp_5_80	0.0002	0.0000	0.0001	0.0001	0.0002

Table 5. Instâncias de Alta Dimensão: Erro relativo médio por instância

Instância	Branch and Bound	Two Approx Algorithm	FPTAS $\varepsilon = 0.8$	FPTAS $\varepsilon = 0.5$	FPTAS $\varepsilon = 0.1$
knapPI_1_10000_1000_1	NA	0.0001	NA	NA	NA
knapPI_1_1000_1000_1	0.0000	0.0021	0.0006	0.0004	0.0001
knapPI_1_100_1000_1	0.0000	0.0361	0.0051	0.0038	0.0006
knapPI_1_2000_1000_1	0.0000	0.0007	NA	NA	NA
knapPI_1_200_1000_1	0.0000	0.0010	0.0028	0.0019	0.0004
knapPI_1_5000_1000_1	NA	0.0003	NA	NA	NA
knapPI_1_500_1000_1	0.0000	0.0008	0.0010	0.0009	0.0002
knapPI_2_10000_1000_1	NA	0.0000	NA	NA	NA
knapPI_2_1000_1000_1	0.0000	0.0007	0.0030	0.0019	NA
knapPI_2_100_1000_1	0.0000	0.0178	0.0273	0.0143	0.0046
knapPI_2_2000_1000_1	0.0000	0.0007	0.0015	0.0011	NA
knapPI_2_200_1000_1	0.0000	0.0184	0.0090	0.0070	0.0014
knapPI_2_5000_1000_1	0.0000	0.0001	NA	NA	NA
knapPI_2_500_1000_1	0.0000	0.0031	0.0049	0.0032	0.0007
knapPI_3_10000_1000_1	NA	0.0002	NA	NA	NA
knapPI_3_1000_1000_1	NA	0.0011	NA	NA	NA
knapPI_3_100_1000_1	0.0000	0.0092	0.0188	0.0183	0.0041
knapPI_3_2000_1000_1	NA	0.0032	NA	NA	NA
knapPI_3_200_1000_1	0.0000	0.0178	0.0108	0.0075	0.0016
knapPI_3_5000_1000_1	NA	0.0008	NA	NA	NA
knapPI_3_500_1000_1	NA	0.0027	0.0057	NA	NA

Table 6. Instâncias de Alta Dimensão: Consumo de memória médio por instância (em bytes)

Instância	Branch and Bound	Two Approx Algorithm	FPTAS $\varepsilon = 0.8$	FPTAS $\varepsilon = 0.5$	FPTAS $\varepsilon = 0.1$
knapPI_1_10000_1000_1	NA	72	NA	NA	NA
knapPI_1_1000_1000_1	462,720	72	14,612,376	23,386,872	116,982,960
knapPI_1_100_1000_1	4,752	72	149,520	239,760	1,203,600
knapPI_1_2000_1000_1	1,187,568	72	NA	NA	NA
knapPI_1_200_1000_1	6,432	72	597,264	956,688	4,793,736
knapPI_1_5000_1000_1	NA	72	NA	NA	NA
knapPI_1_500_1000_1	101,520	72	3,707,664	5,935,656	29,702,016
knapPI_2_10000_1000_1	NA	72	NA	NA	NA
knapPI_2_1000_1000_1	570,192	72	13,955,712	22,335,504	NA
knapPI_2_100_1000_1	2,592	72	146,232	234,648	1,178,352
knapPI_2_2000_1000_1	1,247,568	72	55,805,160	89,301,960	NA
knapPI_2_200_1000_1	6,576	72	574,656	920,880	4,613,928
knapPI_2_5000_1000_1	3,473,424	72	NA	NA	NA
knapPI_2_500_1000_1	138,384	72	3,602,040	5,766,288	28,856,112
knapPI_3_10000_1000_1	NA	72	NA	NA	NA
knapPI_3_1000_1000_1	NA	72	NA	NA	NA
knapPI_3_100_1000_1	14,784	72	168,312	270,000	1,354,848
knapPI_3_2000_1000_1	NA	72	NA	NA	NA
knapPI_3_200_1000_1	44,256	72	656,160	1,051,416	5,266,704
knapPI_3_5000_1000_1	NA	72	NA	NA	NA
knapPI_3_500_1000_1	NA	72	4,151,232	NA	NA

Table 7. Instâncias de Alta Dimensão: Tempo de execução médio por instância (em segundos)

Instância	Branch and Bound	Two Approx Algorithm	FPTAS $\varepsilon = 0.8$	FPTAS $\varepsilon = 0.5$	FPTAS $\varepsilon = 0.1$
knapPI_1_10000_1000_1	NA	2.7873	NA	NA	NA
knapPI_1_1000_1000_1	126.5338	0.0002	302.0068	408.3367	920.0319
knapPI_1_100_1000_1	0.1434	0.0002	0.8528	2.2366	4.6154
knapPI_1_2000_1000_1	415.3066	0.2674	NA	NA	NA
knapPI_1_200_1000_1	0.5985	0.0003	3.5870	5.0264	10.6281
knapPI_1_5000_1000_1	NA	1.0757	NA	NA	NA
knapPI_1_500_1000_1	14.2159	0.0001	50.5120	67.3410	157.1888
knapPI_2_10000_1000_1	NA	2.7851	NA	NA	NA
knapPI_2_1000_1000_1	190.0911	0.0004	371.2476	492.1695	NA
knapPI_2_100_1000_1	0.1721	0.0001	0.3922	0.4878	1.0108
knapPI_2_2000_1000_1	727.5498	0.0009	2827.4479	3850.9851	NA
knapPI_2_200_1000_1	0.5465	0.0001	5.9581	10.1984	34.3029
knapPI_2_5000_1000_1	1481.2707	0.9593	NA	NA	NA
knapPI_2_500_1000_1	46.5634	0.0003	67.5745	89.6811	200.1981
knapPI_3_10000_1000_1	NA	3.0461	NA	NA	NA
knapPI_3_1000_1000_1	NA	0.0700	NA	NA	NA
knapPI_3_100_1000_1	8.1982	0.0001	0.6971	0.6230	2.8097
knapPI_3_2000_1000_1	NA	0.1516	NA	NA	NA
knapPI_3_200_1000_1	183.2889	0.0002	5.2284	7.1143	24.4946
knapPI_3_5000_1000_1	NA	1.4471	NA	NA	NA
knapPI_3_500_1000_1	NA	0.0168	1365.5708	NA	NA

php/389301/mod_folder/content/0/aula15-alg-aproximativos-part3.pdf?forcedownload=1. Acesso em: 06 jul. 2025

[5] LEVITIN, Anany. 2012. Editora Pearson. The Design and Analysis of Algorithms. 3rd edition.

[6] KLEINBERG, Jon; TARDOS, Éva. 2005. Addison-Wesley. *Algorithm Design*. 1st edition.