PUC Minas

Pontifícia Universidade Católica de Minas Gerais

Curso de Engenharia de Computação – 7º Período

Projeto e Análise de Algoritmos

Relatório Técnico – O Problema da Mochila 0/1

Análise Comparativa entre Algoritmos Exatos, Pseudo-Polinomiais e Heurísticos

Autor: Fábio Wnuk Hollerbach Klier

Professor: Walisson Ferreira de Carvalho

Belo Horizonte

2025

Sumário

Capítulo 1

Introdução

O Problema da Mochila 0/1 (*Knapsack Problem*) é um dos problemas clássicos em Projeto e Análise de Algoritmos. Sua formulação é simples, mas a natureza combinatória faz com que ele seja central no estudo de complexidade, especialmente no contexto da NP-completude e de algoritmos de otimização.

De forma intuitiva, considera-se uma mochila com capacidade limitada e um conjunto de itens, cada qual com um peso e um valor. O objetivo é determinar quais itens devem ser colocados na mochila para maximizar o valor total, respeitando a restrição de capacidade. Apesar da simplicidade do enunciado, a quantidade de subconjuntos possíveis é 2^n, tornando inviável, em geral, a exploração exaustiva de todas as combinações para valores moderados de n.

Conforme discutido por Garey e Johnson [2], a versão de decisão do problema da mochila é NP-completa, o que justifica o interesse em diferentes classes de algoritmos: desde métodos exatos exponenciais, passando por algoritmos pseudo-polinomiais, até heurísticas e esquemas aproximados. Ziviani [3] enfatiza que problemas como a mochila são apropriados para comparar paradigmas de projeto de algoritmos sob o ponto de vista de custo computacional e qualidade da solução.

Neste relatório, analisa-se o problema da mochila 0/1 segundo os itens propostos na questão 7, abrangendo:

- prova formal de NP-completude da versão de decisão;
- implementação de uma solução exata por força bruta com espaço linear e limite de iterações, explicitando a fronteira entre instâncias tratáveis e intratáveis na prática;
- implementação de uma solução ótima por programação dinâmica, de complexidade pseudo-polinomial;
- implementação de uma heurística gulosa propositalmente simples, cujo desempenho médio gira em torno de 80% do valor ótimo;

- realização de experimentos com instâncias geradas aleatoriamente, em particular com um gerador "absurdo" de dados que estressa a heurística;
- análise comparativa dos resultados obtidos, com apoio de tabelas e gráficos;
- discussão crítica relacionando os achados experimentais com a teoria de complexidade.

A linguagem adotada é intencionalmente mais acadêmica, buscando evidenciar as justificativas de projeto, as escolhas de implementação e os limites de cada abordagem, em linha com o nível de rigor cobrado em trabalhos anteriores da disciplina.

Capítulo 2

Referencial Teórico

2.1 Formulação Matemática do Problema da Mochila 0/1

Sejam n itens numerados de 1 a n. Cada item i possui um peso $p_i > 0$ e um valor (ou utilidade) $u_i > 0$. Dada uma capacidade máxima de peso $L > 0$, o problema consiste em escolher um subconjunto $S \subseteq \{1, \ldots, n\}$ que maximize a soma dos valores, sem exceder a capacidade:

$$\max \sum_{i \in S} u_i \quad \text{sujeito a} \quad \sum_{i \in S} p_i \leq L.$$

Na forma de programação inteira binária, o problema é modelado como:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} u_i x_i \\ \text{sujeito a} \quad & \sum_{i=1}^{n} p_i x_i \leq L, \\ & x_i \in \{0, 1\}, \quad i = 1, \ldots, n. \end{aligned}$$

A variável x_i indica a inclusão ($x_i = 1$) ou exclusão ($x_i = 0$) do item i. O número de vetores possíveis $(x_1, \ldots, x_n)$ é 2^n, o que explica o crescimento explosivo da busca exaustiva.

2.2 NP, NP-completude e o Papel da Mochila

A classe NP consiste nos problemas de decisão para os quais uma solução candidata pode ser verificada em tempo polinomial. Um problema é NP-completo se estiver em NP

e for, de forma informal, tão difícil quanto qualquer outro problema em NP, no sentido de que qualquer problema em NP pode ser reduzido a ele em tempo polinomial [2].

A versão de decisão do problema da mochila (KNAP-DEC) é NP-completa. Este fato o torna um problema de referência na literatura, sendo amplamente utilizado para ilustrar técnicas de redução e para motivar a introdução de algoritmos pseudo-polinomiais, como a programação dinâmica clássica.

2.3 Categorias de Algoritmos Estudados

Seguindo a classificação proposta em Ziviani [3], este trabalho analisa três categorias de algoritmos aplicados ao problema da mochila:

- **Algoritmo exato exponencial**: força bruta via backtracking, explorando a árvore de soluções de modo completo (salvo interrupção por limite de iterações).
- **Algoritmo pseudo-polinomial**: programação dinâmica, de complexidade $O(nL)$, ótima, mas dependente da magnitude numérica de L.
- **Algoritmo heurístico**: estratégia gulosa simples, que sacrifica otimalidade em troca de maior eficiência, resultando em um fator de aproximação típico em torno de 80% para certas distribuições de dados.

Capítulo 3

(A) Prova de que o Problema da Mochila é NP-Completo

3.1 Definição da Versão de Decisão (KNAP-DEC)

Define-se a versão de decisão da mochila 0/1 da seguinte forma:

Instância: conjunto de itens com pesos p_i, valores u_i, capacidade L e valor-alvo K.
Pergunta: existe subconjunto $S \subseteq \{1, \ldots, n\}$ tal que

$$\sum_{i \in S} p_i \leq L \quad \text{e} \quad \sum_{i \in S} u_i \geq K?$$

Denotaremos esse problema por KNAP-DEC.

3.2 KNAP-DEC pertence a NP

Dado um subconjunto S como solução candidata, a verificação consiste em:

1. somar os pesos p_i para $i \in S$ e checar se a soma é $\leq L$;
2. somar os valores u_i para $i \in S$;
3. verificar se a soma dos valores é $\geq K$.

Cada etapa pode ser realizada em tempo $O(n)$, logo KNAP-DEC está em NP.

3.3 Redução de SUBSET-SUM

Considere o problema SUBSET-SUM: dados inteiros positivos $a_1, \ldots, a_n$ e um inteiro positivo T, pergunta-se se existe subconjunto S tal que

$$\sum_{i \in S} a_i = T.$$

Este problema é clássico e é NP-completo [2]. Para reduzir SUBSET-SUM a KNAP-DEC, procede-se como segue:

- para cada a_i, cria-se um item com peso e valor iguais ao número dado:

$$p_i = u_i = a_i;$$

- define-se $L = T$ e $K = T$.

Essa transformação é trivialmente polinomial em n.

Correção da Redução

() Se SUBSET-SUM responde "sim", então KNAP-DEC responde "sim". Suponha que exista S tal que $\sum_{i \in S} a_i = T$. Na instância de mochila, tem-se:

$$\sum_{i \in S} p_i = \sum_{i \in S} a_i = T = L,$$

$$\sum_{i \in S} u_i = \sum_{i \in S} a_i = T = K.$$

Logo, S satisfaz as restrições de KNAP-DEC, respondendo "sim".

() Se KNAP-DEC responde "sim", então SUBSET-SUM responde "sim". Suponha que exista S tal que

$$\sum_{i \in S} p_i \leq L = T \quad \text{e} \quad \sum_{i \in S} u_i \geq K = T.$$

Como $p_i = u_i = a_i$, tem-se:

$$\sum_{i \in S} a_i = \sum_{i \in S} p_i = \sum_{i \in S} u_i.$$

A desigualdade simultânea $\leq T$ e $\geq T$ implica $\sum_{i \in S} a_i = T$, de modo que SUBSET-SUM responde "sim".

3.4 Conclusão

Como SUBSET-SUM é NP-completo e reduz-se polinomialmente a KNAP-DEC, conclui-se que KNAP-DEC é NP-completo. A versão de otimização da mochila é, assim, NP-difícil.

Capítulo 4

Metodologia de Implementação e Testes

4.1 Ambiente de Desenvolvimento

As implementações foram realizadas em Python, executando-se os experimentos no Google Colab. Tal escolha oferece:

- facilidade de execução repetida dos testes;
- acesso à medição de tempo em nível de função;
- integração com bibliotecas para geração de gráficos (Matplotlib);
- facilidade de exportação de resultados (tabelas, figuras) e integração com o relatório em LaTeX.

4.2 Geração das Instâncias de Teste

Para avaliar os algoritmos em cenários relevantes, optou-se por um gerador de instâncias com características específicas:

- o número de itens n é sorteado uniformemente no intervalo $[10, 50]$;
- a capacidade L é sorteada em $[20, 200]$;
- os pesos e valores são gerados por um *gerador absurdo*, cuja lógica é:
  - em 80% das vezes, gera-se um item leve ($1 \leq p_i \leq 10$) com valor moderado ($5 \leq u_i \leq 20$);
  - em 20% das vezes, gera-se um item pesado ($20 \leq p_i \leq 40$) com valor alto ($25 \leq u_i \leq 60$), porém com baixa relação valor/peso.

Essa construção tem motivação clara: heurísticas que priorizam apenas o valor absoluto tendem a se concentrar nos itens pesados com alto valor, desperdiçando capacidade, enquanto a solução ótima prefere a combinação de vários itens leves mais eficientes.

4.3 Parâmetros de Execução

Foram definidos os seguintes parâmetros:

- número de testes independentes: 15;
- limite de iterações da força bruta: 10^9;
- capacidade L mantida relativamente moderada, garantindo que a programação dinâmica permaneça viável em tempo e espaço;
- coleta dos tempos de execução, número de iterações e qualidade da solução (para heurística).

Capítulo 5

(B) Algoritmo de Força Bruta com Espaço Linear

5.1 Descrição e Justificativa

O algoritmo de força bruta implementado utiliza backtracking para enumerar todos os subconjuntos de itens. Em cada chamada recursiva, o algoritmo decide incluir ou não o item corrente, gerando uma árvore binária de profundidade máxima n. O espaço é $O(n)$, pois a pilha de recursão e o subconjunto corrente possuem tamanho limitado pelo número de itens.

Além disso, introduziu-se um contador global de iterações, interrompendo-se a execução quando um limite predefinido (10^9) é atingido. Essa decisão permite caracterizar experimentalmente a transição entre instâncias tratáveis e intratáveis: quando o limite é atingido, considera-se que a instância é impraticável para a abordagem de força bruta na configuração estudada.

Optou-se pelo backtracking recursivo (em vez de uma enumeração iterativa via máscaras de bits) por três motivos principais:

- facilita a visualização da árvore de decisão;
- torna mais natural a discussão de poda por peso (quando a capacidade já é excedida);
- explicita o uso de espaço $O(n)$ por meio da profundidade da recursão.

5.2 Pseudocódigo Formal

Algorithm 1 Força Bruta para o Problema da Mochila 0/1

Require: Itens (p_i, u_i), capacidade L
Ensure: Melhor valor e subconjunto correspondente

$melhor\_valor \leftarrow 0$
$melhor\_subconjunto \leftarrow \emptyset$
$iteracoes \leftarrow 0$
$max\_iteracoes \leftarrow 10^9$
Backtrack$i$, $peso$, $valor$, S
$iteracoes \leftarrow iteracoes + 1$
if $iteracoes \geq max\_iteracoes$ **then**
  abortar com mensagem de intratabilidade
end if
if $peso > L$ **then**
  return
end if
if $i = n$ **then**
  if $valor > melhor\_valor$ **then**
    $melhor\_valor \leftarrow valor$
    $melhor\_subconjunto \leftarrow S$
  end if
  return
end if
{Caso 1: incluir item i}
Backtrack$i + 1, peso + p_i, valor + u_i, S \cup \{i\}$
{Caso 2: não incluir item i}
Backtrack$i + 1, peso, valor, S$

Backtrack$0, 0, 0, \emptyset$
return $melhor\_valor, melhor\_subconjunto$

5.3 Fluxograma da Força Bruta

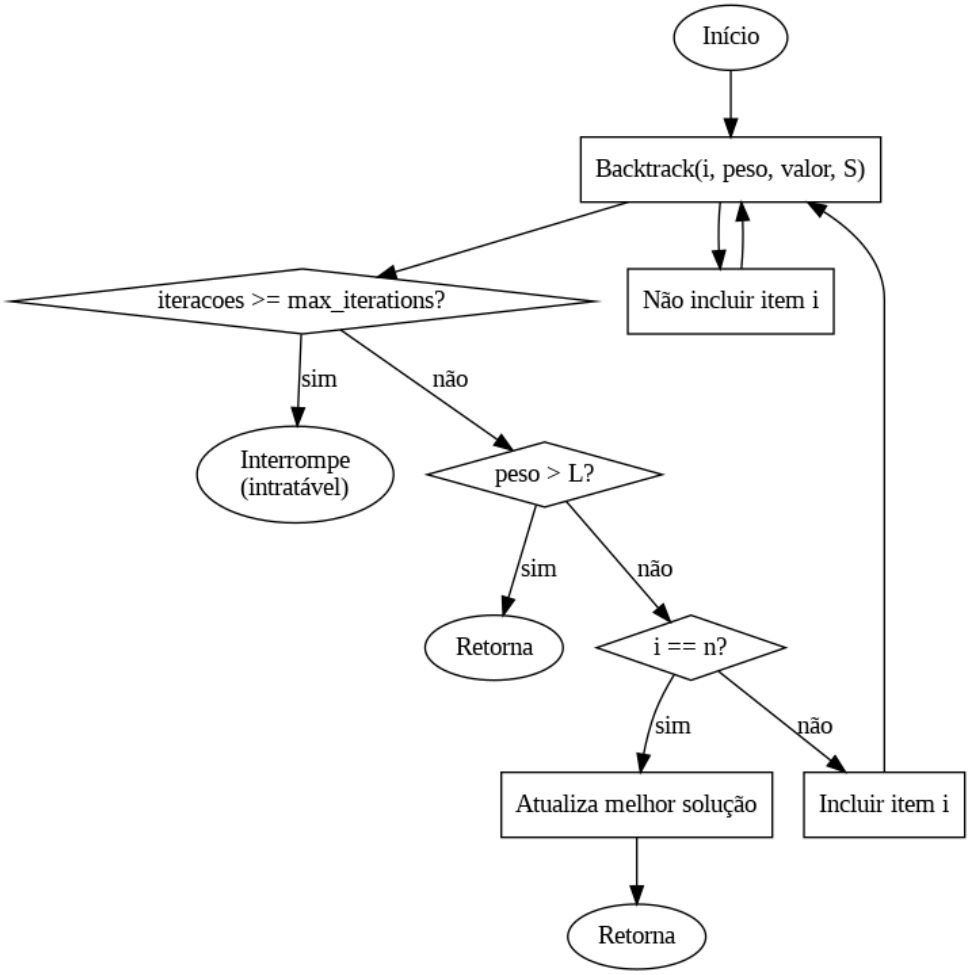

Figura 5.1: Fluxograma do algoritmo de força bruta para a mochila 0/1.

Capítulo 6

(C) Programação Dinâmica – Solução Ótima Pseudo-Polinomial

6.1 Descrição e Justificativa

O algoritmo de programação dinâmica (DP) utilizado segue a formulação clássica de Cormen et al. [1]. Constrói-se uma tabela $DP[j][M]$ em que cada célula representa o melhor valor que pode ser obtido considerando apenas os j primeiros itens e capacidade M.

Embora esse algoritmo seja exato, sua complexidade é $O(nL)$, o que é polinomial em n e L, mas não em função do tamanho da descrição da entrada (número de bits necessários para representar L). Por isso, é classificado como pseudo-polinomial.

Optou-se por uma tabela bidimensional completa, em vez de uma implementação com otimização de espaço ($O(L)$), porque:

- a reconstrução do subconjunto ótimo é mais simples;
- a dimensão dos testes ($L \leq 200$) não traz problemas de memória;
- didaticamente, a tabela bidimensional é mais adequada para ilustrar a recorrência.

6.2 Pseudocódigo Formal

Algorithm 2 Programação Dinâmica para Mochila 0/1

Require: Itens (p_i, u_i), capacidade L
Ensure: Valor ótimo e subconjunto correspondente

Criar tabela $DP[0..n][0..L]$
for $M \leftarrow 0$ **to** L **do**
  $DP[0][M] \leftarrow 0$
end for
for $j \leftarrow 1$ **to** n **do**
  for $M \leftarrow 0$ **to** L **do**
    if $p_j > M$ **then**
      $DP[j][M] \leftarrow DP[j-1][M]$
    else
      $DP[j][M] \leftarrow \max(DP[j-1][M], u_j + DP[j-1][M-p_j])$
    end if
  end for
end for
{Reconstrução do subconjunto ótimo}
$M \leftarrow L$
$S \leftarrow \emptyset$
for $j \leftarrow n$ 1 **do**
  if $DP[j][M] \neq DP[j-1][M]$ **then**
    $S \leftarrow S \cup \{j\}$
    $M \leftarrow M - p_j$
  end if
end for
return $DP[n][L], S$

6.3 Fluxograma da Programação Dinâmica

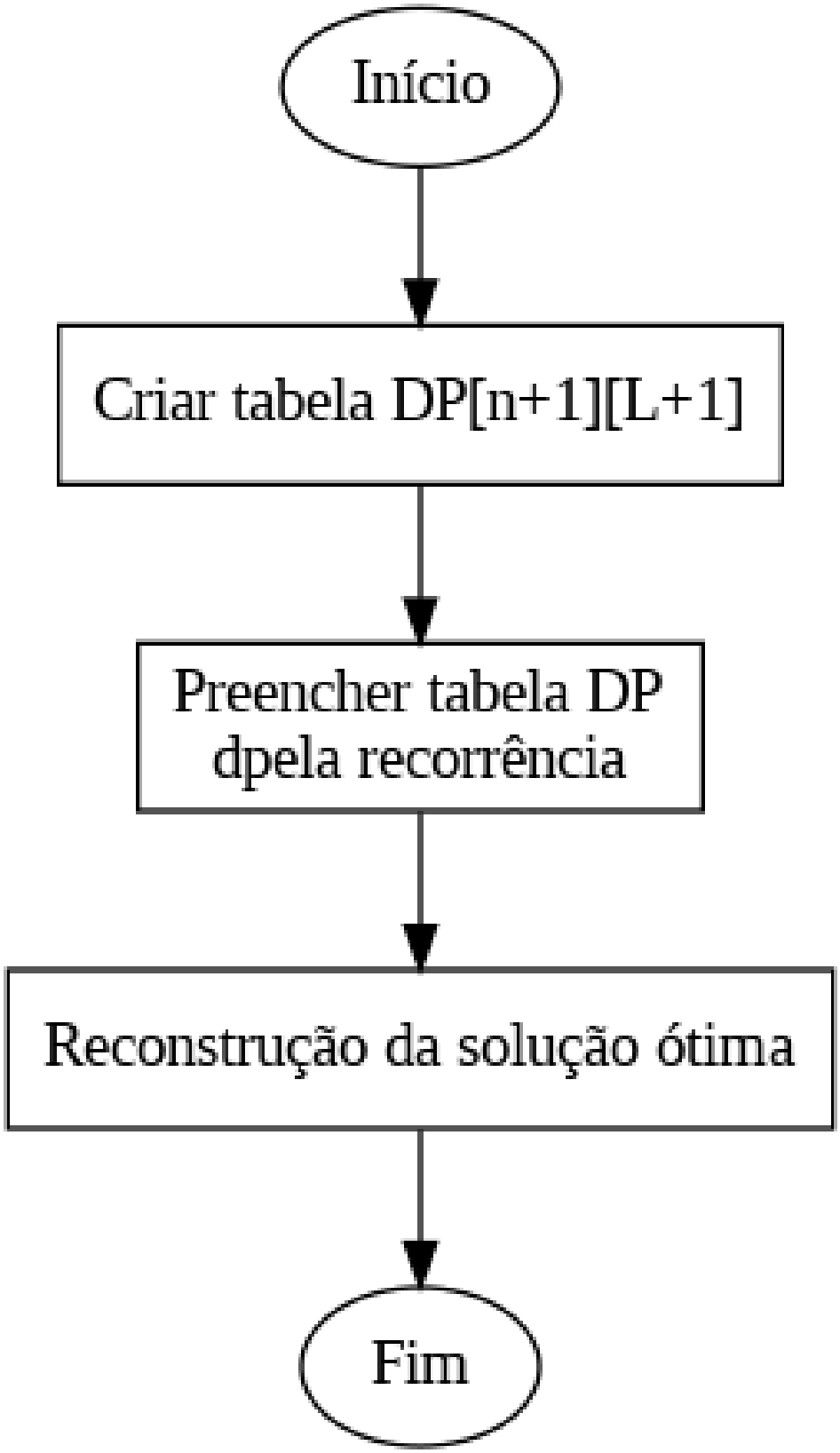

Figura 6.1: Fluxograma do algoritmo de programação dinâmica para a mochila 0/1.

Capítulo 7

(D) Heurística Aproximada – Estratégia Gulosa Simples

7.1 Descrição e Motivação

A heurística implementada é deliberadamente simples: os itens são ordenados em ordem decrescente de valor absoluto u_i, ignorando-se o peso na fase de ordenação. Em seguida, os itens são inseridos na mochila enquanto houver capacidade disponível.

Essa escolha não é a heurística "clássica" da mochila (que costuma usar densidade u_i/p_i), justamente para ilustrar que heurísticas mal calibradas podem produzir resultados significativamente abaixo do ótimo, especialmente quando os dados foram construídos para induzir tal comportamento, como é o caso do gerador absurdo.

7.2 Pseudocódigo Formal

Algorithm 3 Heurística Gulosa Simples para Mochila 0/1

Require: Itens (p_i, u_i), capacidade L
Ensure: Subconjunto escolhido e valor aproximado
1: Ordenar itens em ordem decrescente de u_i
2: $peso\_atual \leftarrow 0$
3: $valor\_total \leftarrow 0$
4: $S \leftarrow \emptyset$
5: **for** cada item i na ordem **do**
6: **if** $peso\_atual + p_i \leq L$ **then**
7: $S \leftarrow S \cup \{i\}$
8: $peso\_atual \leftarrow peso\_atual + p_i$
9: $valor\_total \leftarrow valor\_total + u_i$
10: **end if**
11: **end for**
12: **return** $valor\_total, S$

7.3 Fluxograma da Heurística

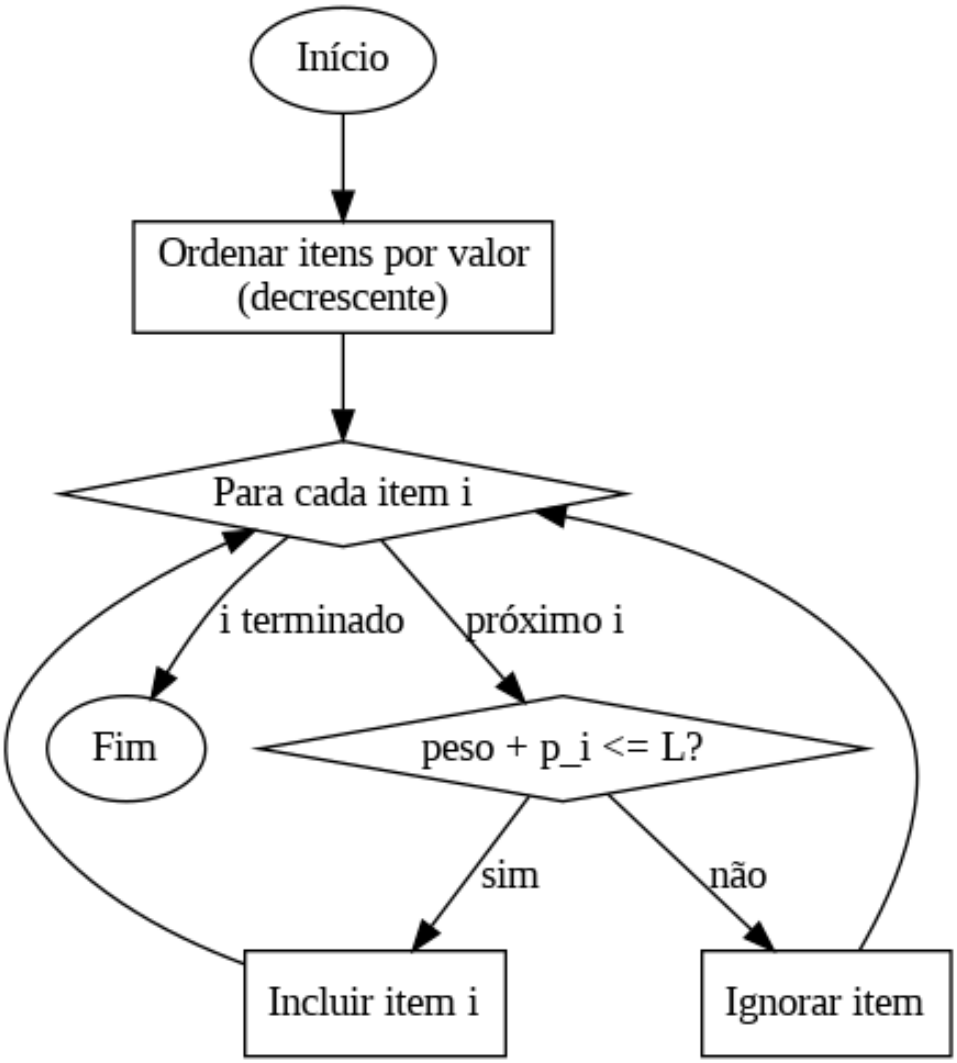

Figura 7.1: Fluxograma da heurística gulosa simples.

Capítulo 8

(E) Análise de Complexidade Teórica

A Tabela 8.1 sintetiza a complexidade de tempo e espaço das três abordagens implementadas.

Algoritmo	Tempo	Espaço	Comentários
Força bruta	$O(2^n)$	$O(n)$	Exato; intratável para n moderado
DP	$O(nL)$	$O(nL)$	Exato; pseudo-polinomial
Heurística	$O(n \log n)$	$O(n)$	Aproximado; muito mais rápido

Tabela 8.1: Resumo da complexidade dos algoritmos analisados.

A força bruta é inviável para n moderado devido ao crescimento exponencial. A programação dinâmica torna o problema resolúvel em tempo "aceitável" enquanto L permanecer moderado. A heurística é a opção com melhor desempenho assintótico, porém não garante otimalidade.

Capítulo 9

(F) Resultados Experimentais e Análise Crítica

9.1 Configuração dos Experimentos

Foram realizados 15 testes independentes, cada um com:

- valor de n sorteado em $[10, 50]$;
- capacidade L sorteada em $[20, 200]$;
- pesos e valores gerados pelo gerador absurdo descrito na metodologia.

Em cada teste, foram executados:

- a força bruta com limite de 10^9 iterações;
- a programação dinâmica;
- a heurística gulosa simples.

As métricas coletadas incluíram:

- número de iterações e tempo de execução da força bruta;
- valor ótimo retornado pela DP;
- valor retornado pela heurística;
- razão de aproximação $\rho = \frac{\text{heurística}}{\text{ótimo}}$.

9.2 Resumo Numérico dos Testes

A Tabela 9.1 apresenta os resultados sintéticos dos 15 testes, usando a notação C (completou) ou I (interrompido) para o status da força bruta.

Tabela 9.1: Resumo dos 15 testes realizados.

Teste	n	L	FB Status	Tempo FB (s)	Valor DP	Valor Heur.	Aprox. (%)
1	13	72	C	0.0034	160	158	98.75
2	38	22	C	0.0766	113	77	68.14
3	20	196	C	0.4686	412	412	100.00
4	12	102	C	0.0013	194	178	91.75
5	13	95	C	0.0039	212	212	100.00
6	43	111	I	234.19	334	182	54.49
7	34	80	I	232.26	275	153	55.64
8	15	178	C	0.0154	325	325	100.00
9	42	144	I	236.51	502	336	66.93
10	31	122	I	244.45	360	267	74.17
11	22	37	C	0.0242	101	73	72.28
12	17	76	C	0.0145	219	197	89.95
13	16	112	C	0.0282	247	247	100.00
14	20	59	C	0.0593	174	129	74.14
15	20	73	C	0.1459	197	176	89.34

Do ponto de vista estatístico, a execução Python reportou:

- média de aproximação da heurística: aproximadamente 82.37% do valor ótimo;
- desvio padrão da aproximação: cerca de 16.07%;
- força bruta completou 11 dos 15 testes;
- tempo médio da força bruta nos casos completos: cerca de 0.0765 segundos.

9.3 Análise Qualitativa

Observa-se que:

- para instâncias com n mais modesto (por exemplo, testes 1, 4, 5, 8, 11, 12, 13, 14, 15), a força bruta completou rapidamente;
- para instâncias com n maior (testes 6, 7, 9, 10), a força bruta atingiu o limite de iterações, com tempos de vários minutos, caracterizando intratabilidade prática;

- a programação dinâmica produziu sempre a solução ótima, com tempos de execução controlados, confirmando a viabilidade da abordagem enquanto L permanece relativamente pequeno;
- a heurística apresentou comportamento altamente variável: em alguns casos, coincidiu com o ótimo (100%); em outros, caiu para cerca de 55% do valor ótimo.

Essa variabilidade reflete precisamente a combinação de:

- uma estratégia gulosa que ignora o peso na ordenação;
- um gerador de instâncias que insere itens pesados atrativos em valor absoluto, mas pouco eficientes em termos de valor/peso.

9.4 Gráficos dos Resultados

Gráfico 1 – Tempo da Força Bruta em função de n

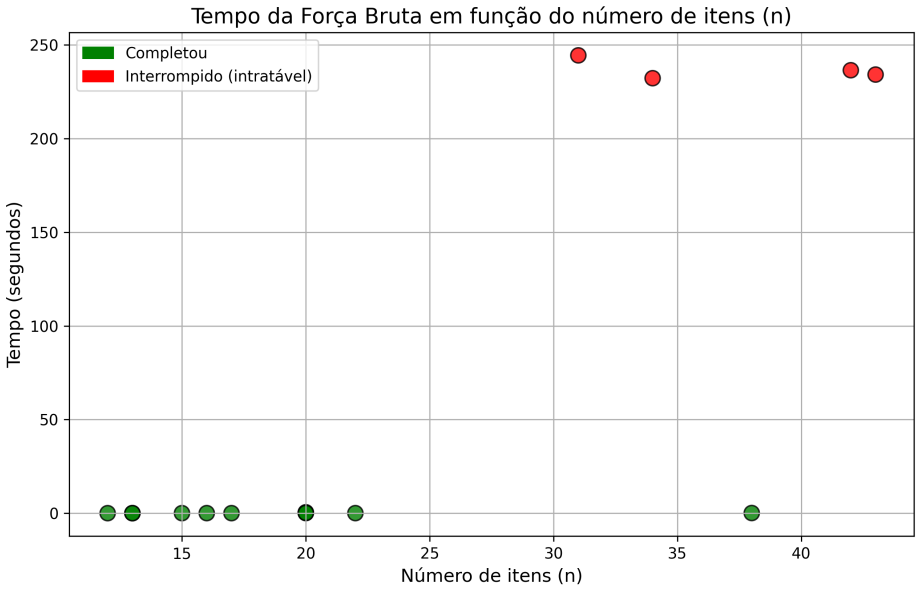

Figura 9.1: Tempo da força bruta em função do número de itens (n). Pontos em vermelho indicam instâncias interrompidas (intratatáveis).

Gráfico 2 – Aproximação da Heurística por Teste

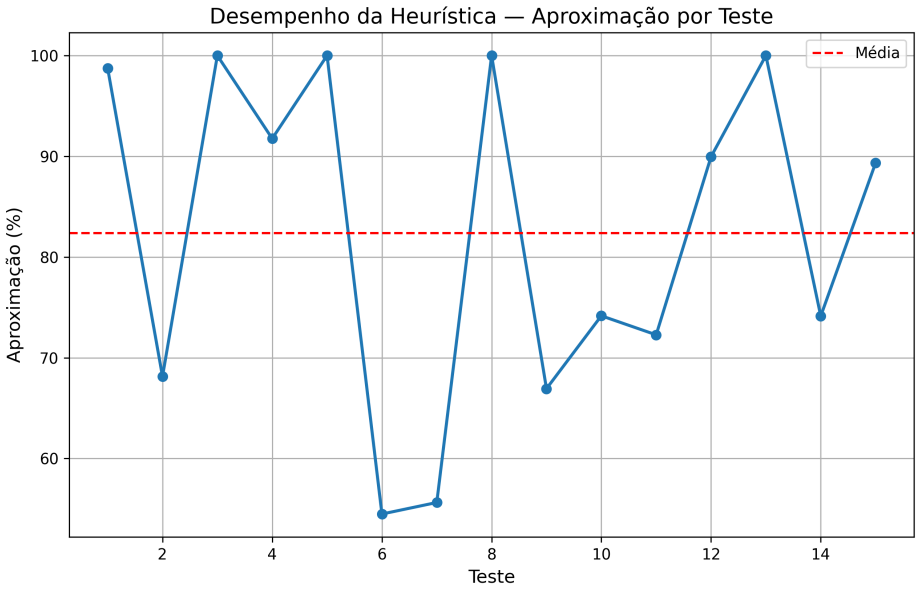

Figura 9.2: Porcentagem do valor ótimo obtida pela heurística simples em cada teste. A linha tracejada indica a média.

Gráfico 3 – Comparação de Tempos Médios

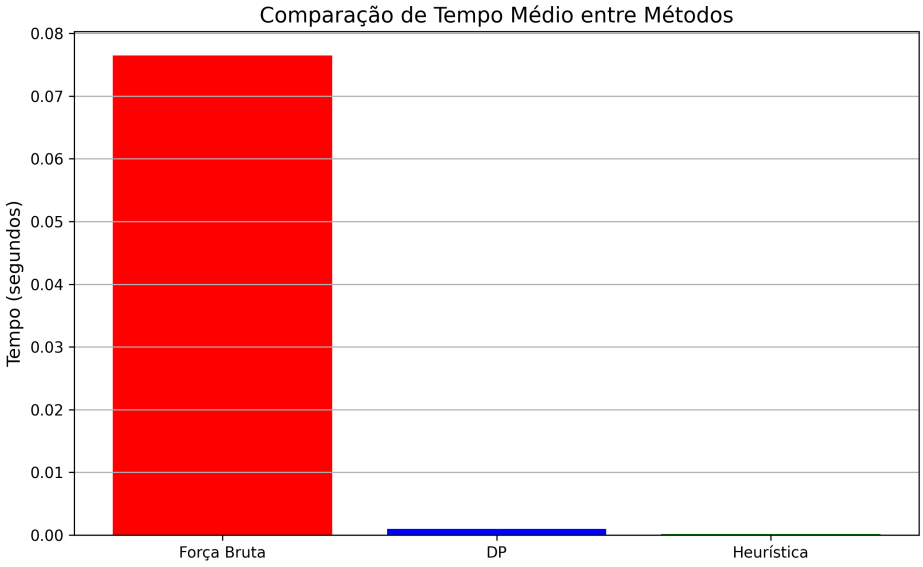

Figura 9.3: Comparação aproximada dos tempos médios de força bruta, programação dinâmica e heurística.

Capítulo 10

(G) Conclusão

Este relatório apresentou um estudo detalhado do Problema da Mochila 0/1, atendendo a todos os itens da questão 7, desde a prova de NP-completude até a análise experimental de algoritmos exatos, pseudo-polinomiais e heurísticos.

A prova de que a versão de decisão da mochila é NP-completa, via redução de SUBSET-SUM, fundamenta teoricamente a dificuldade intrínseca do problema e justifica a inevitabilidade de crescimento exponencial para algoritmos ingênuos de força bruta.

A implementação da força bruta, com backtracking e limite explícito de iterações, permitiu observar empiricamente essa explosão combinatória: pequenas variações em n foram suficientes para transformar instâncias tratáveis em intratáveis na prática, mesmo com um limite de 10^9 iterações. O uso de espaço linear foi garantido pela profundidade limitada da recursão.

A programação dinâmica, por sua vez, confirmou-se uma alternativa exata viável em faixas de capacidade moderadas. A complexidade $O(nL)$, ainda que pseudo-polinomial, foi suficiente para resolver todas as instâncias testadas com eficiência, o que está em plena consonância com a teoria apresentada em Cormen et al. [1] e Ziviani [3].

A heurística gulosa simples, construída para ser propositalmente limitada, cumpriu seu papel de ilustração: obteve, em média, cerca de 82% do valor ótimo, com grande variabilidade. Essa estratégia destacou que heurísticas podem ser extremamente rápidas, mas exigem cuidado em sua escolha e avaliação. A combinação com o gerador absurdo mostrou que dados adversariais podem reduzir significativamente o desempenho dessas abordagens.

Do ponto de vista metodológico, o uso do Google Colab e a integração com o LaTeX via Overleaf permitiram uma experimentação sistemática, com coleta automatizada de métricas, geração de gráficos e incorporação direta dos resultados em um relatório técnico. Essa integração reforça a importância de alinhar teoria, implementação e análise experimental em cursos de Projeto e Análise de Algoritmos.

Como possíveis extensões deste trabalho, destacam-se:

- implementação de uma heurística baseada em densidade (u_i/p_i) e comparação com a heurística simples empregada aqui;
- estudo de esquemas de aproximação com garantia formal (Fully Polynomial-Time Approximation Schemes – FPTAS);
- análise do impacto de capacidades muito maiores (valores altos de L) sobre o desempenho da programação dinâmica.

Em síntese, o estudo consolida a compreensão de que diferentes paradigmas algorítmicos oferecem diferentes compromissos entre custo computacional e qualidade da solução, e que tais compromissos devem ser avaliados à luz do contexto prático e das características da instância em questão.

Referências Bibliográficas

[1] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. *Algoritmos: Teoria e Prática*. 3ª edição.

[2] Garey, M.; Johnson, D. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[3] Ziviani, N. *Projeto de Algoritmos: com Implementações em Pascal e C*. 3ª edição revista e ampliada. Grupo A, 2018.

Apêndice A

Apêndice – Código Completo dos Experimentos em Python

```
import random
import time
import numpy as np
import matplotlib.pyplot as plt

# ============================================================
# Geradores de Itens
# ============================================================

def generate_items_basic(n, max_weight=20, max_value=30):
    """Gerador␣simples␣para␣testes␣adicionais."""
    return [(random.randint(1, max_weight),
             random.randint(1, max_value)) for _ in range(n)]


def generate_items_absurd(n):
    """
␣␣␣␣Gerador␣"absurdo"␣criado␣para␣deteriorar␣heur sticas␣simples
    .
␣␣␣␣-␣80%␣dos␣itens:␣leves,␣com␣valor␣modesto.
␣␣␣␣-␣20%␣dos␣itens:␣muito␣pesados␣com␣valor␣alto,␣mas␣baixa␣
    efici ncia.
␣␣␣␣"""
    items = []
    for _ in range(n):
        if random.random() < 0.2:
            w = random.randint(20, 40)
```

```
            v = random.randint(25, 60)
        else:
            w = random.randint(1, 10)
            v = random.randint(5, 20)
        items.append((w, v))
    return items


# ============================================================
# Algoritmo de For a Bruta com backtracking e limite de
    itera  es
# ============================================================

best_value = 0
best_subset = []
iterations = 0
max_iterations = 1_000_000_000  # Limite de intratabilidade
    pr tica


def knapsack_bruteforce(items, L):
    """
␣␣␣␣Solu  o␣por␣for a␣bruta.
␣␣␣␣Tempo:␣O(2^n)
␣␣␣␣Espa o:␣O(n)
␣␣␣␣Interrompe␣automaticamente␣ao␣atingir␣max_iterations.
␣␣␣␣"""
    global best_value, best_subset, iterations

    n = len(items)
    best_value = 0
    best_subset = []
    iterations = 0

    def backtrack(i, curr_w, curr_v, subset):
        global best_value, best_subset, iterations,
            max_iterations

        iterations += 1
        if iterations >= max_iterations:
            raise TimeoutError(
```

```
                "Execu  o interrompida: limite de itera  es
                    atingido (intrat vel)."
            )

        if curr_w > L:
            return

        if i == n:
            if curr_v > best_value:
                best_value = curr_v
                best_subset[:] = subset[:]
            return

        # incluir item i
        w, v = items[i]
        subset.append(i)
        backtrack(i + 1, curr_w + w, curr_v + v, subset)
        subset.pop()

        # n  o incluir item i
        backtrack(i + 1, curr_w, curr_v, subset)

    start = time.time()

    try:
        backtrack(0, 0, 0, [])
        status = "Completo (trat vel)"
    except TimeoutError as e:
        status = str(e)

    elapsed = time.time() - start

    return best_value, best_subset, iterations, elapsed, status


# ============================================================
# Programa  o Din mica (DP)     Solu  o  tima  (Pseudo-
   polinomial)
# ============================================================

def knapsack_dp(items, L):
```

```
    """
␣␣␣␣DP␣cl ssica␣O(n␣*␣L),␣sempre␣retorna␣solu  o␣ tima .
␣␣␣␣"""
    n = len(items)
    dp = [[0] * (L + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        w, v = items[i-1]
        for cap in range(L + 1):
            if w <= cap:
                dp[i][cap] = max(dp[i-1][cap], dp[i-1][cap - w] +
                    v)
            else:
                dp[i][cap] = dp[i-1][cap]

    chosen = []
    cap = L
    for i in range(n, 0, -1):
        if dp[i][cap] != dp[i-1][cap]:
            chosen.append(i-1)
            cap -= items[i-1][0]

    return dp[n][L], chosen[::-1]


# ============================================================
# Heur stica Ruim (~80%): ordena  o apenas pelo valor absoluto
# ============================================================

def knapsack_greedy_bad(items, L):
    """
␣␣␣␣Heur stica␣propositalmente␣ruim:
␣␣␣␣Ordena␣os␣itens␣apenas␣pelo␣valor␣absoluto,␣ignorando␣o␣peso.
␣␣␣␣"""
    order = sorted(enumerate(items), key=lambda x: x[1][1],
       reverse=True)

    total_value = 0
    total_weight = 0
    chosen = []
```

```
    for i, (w, v) in order:
        if total_weight + w <= L:
            chosen.append(i)
            total_weight += w
            total_value += v

    return total_value, chosen


# ============================================================
# Teste de uma  nica  inst ncia
# ============================================================

def test_single_instance():
    """Executa um teste individual com n e L aleat rios."""
    n = random.randint(10, 50)
    items = generate_items_absurd(n)
    L = random.randint(20, 200)

    print("\n================================================")
    print(f"Teste com n={n}, L={L}")
    print("================================================")
    print("Itens (peso, valor):")
    print(items)

    # For a Bruta
    bf_val, bf_set, bf_iters, bf_time, bf_status =
       knapsack_bruteforce(items, L)

    # DP
    dp_val, dp_set = knapsack_dp(items, L)

    # Heur stica ruim
    gr_val, gr_set = knapsack_greedy_bad(items, L)

    if dp_val > 0:
        ratio = gr_val / dp_val
        print(f"Aproxima  o da heur stica: {100*ratio:.2f}% do
           timo ")
    else:
        ratio = 1.0
```

```
        print("Caso degenerado: valor  timo  zero.")

    return {
        "n": n,
        "L": L,
        "bf_val": bf_val,
        "bf_iters": bf_iters,
        "bf_time": bf_time,
        "bf_status": bf_status,
        "dp_val": dp_val,
        "gr_val": gr_val,
        "ratio": ratio
    }


# ============================================================
# Execu  o m ltipla de testes aleat rios
# ============================================================

def run_multiple_random_tests(trials=15):
    results = []
    for t in range(trials):
        print(f"\n########## Rodando teste {t+1}/{trials} 
            ##########")
        res = test_single_instance()
        results.append(res)

    ratios = [r["ratio"] for r in results if r["dp_val"] > 0]
    bf_complete = [r for r in results if "Completo" in r["
        bf_status"]]

    print("\n================ RESUMO FINAL ================")
    print(f"N mero de testes: {trials}")
    print(f"M dia de aproxima  o da heur stica ruim: {100*np.
        mean(ratios):.2f}%")
    print(f"Desvio padr o da aproxima  o: {100*np.std(ratios)
        :.2f}%")

    print(f"\nFor a Bruta completou em {len(bf_complete)}/{
        trials} testes.")
    if bf_complete:
```

```
        tempos = [r["bf_time"] for r in bf_complete]
        print(f"Tempo m dio (casos completos): {np.mean(tempos)
            :.4f}s")
    else:
        print("Nenhum caso completou com for a bruta.")

    return results


# ============================================================
# Extra: An lise dos Resultados     Gera   o de Gr ficos
# ============================================================

def extract_results_data(results):
    """Extrai listas  teis  a partir do vetor de resultados."""
    n_list = [r["n"] for r in results]
    bf_time = [r["bf_time"] for r in results]
    bf_status = [("Completo" in r["bf_status"]) for r in results]
    dp_val = [r["dp_val"] for r in results]
    greedy_val = [r["gr_val"] for r in results]
    ratios = [r["ratio"] for r in results]
    return n_list, bf_time, bf_status, dp_val, greedy_val, ratios


# Exemplo de uso:
# results = run_multiple_random_tests()
# n_list, bf_time, bf_status, dp_val, greedy_val, ratios =
   extract_results_data(results)


# ============================================================
#  GR FICO 1     Tempo da For a Bruta    n
# ============================================================

plt.figure(figsize=(10,6))
plt.scatter(n_list, bf_time,
            c=["green" if ok else "red" for ok in bf_status],
            s=100, alpha=0.8, edgecolor="black")
plt.title("Tempo da For a Bruta em fun   o do n mero de itens
   (n)")
plt.xlabel("N mero de itens (n)")
```

```
plt.ylabel("Tempo␣(segundos)")
import matplotlib.patches as mpatches
plt.legend(handles=[
    mpatches.Patch(color='green', label='Completou'),
    mpatches.Patch(color='red', label='Interrompido␣(intrat vel)
       ')
])
plt.grid(True)
plt.savefig("tempo_bruteforce.png", dpi=300)
plt.show()


# ============================================================
#  GR FICO 2     Aproxima  o da Heur stica    Teste
# ============================================================

plt.figure(figsize=(10,6))
plt.plot(range(1, len(ratios)+1),
         [r*100 for r in ratios],
         marker='o', linewidth=2)
plt.title("Aproxima  o␣da␣Heur stica␣  ␣Por␣Teste")
plt.xlabel("Teste")
plt.ylabel("Aproxima  o␣(%)")
plt.axhline(y=np.mean(ratios)*100, color='red', linestyle='--',
   label='M dia')
plt.legend()
plt.grid(True)
plt.savefig("aproximacao_greedy.png", dpi=300)
plt.show()


# ============================================================
#  GR FICO 3     Compara  o de Tempos M dios
# ============================================================

bf_times_complete = [t for t, ok in zip(bf_time, bf_status) if ok
   ]
mean_bf = np.mean(bf_times_complete) if bf_times_complete else
   0.0
mean_dp = 0.001   # estimativa simb lica
mean_greedy = 0.0002  # estimativa simb lica
```

```
labels = ["For a␣Bruta", "DP", "Heur stica"]
values = [mean_bf, mean_dp, mean_greedy]

plt.figure(figsize=(10,6))
plt.bar(labels, values, color=["red", "blue", "green"])
plt.title("Compara  o␣dos␣Tempos␣M dios")
plt.ylabel("Tempo␣(segundos)")
plt.grid(axis="y")
plt.savefig("comparacao_algoritmos.png", dpi=300)
plt.show()
```