Análise e Implementação de Algoritmos para o Problema da Mochila 0/1

Eduardo Henrique de Almeida Izidorio[1], Shelly da Costa Leal[2]

[1]Departamento de Ciência da Computação – Universidade Federal de Roraima

Campus Paricarana, Av. Cap. Ene Garcês, 2413 – Boa Vista – RR, 69310-000

{eduardo57izidorio, contatoshellyleal05}@gmail.com

Resumo. *Este trabalho apresenta a análise do artigo The Price-Elastic Knapsack Problem (Fukasawa et al., 2024), publicado em periódico indexado no IEEE, que aborda uma variação do Problema da Mochila 0/1 incorporando elasticidade de preços, tornando-o mais próximo de cenários reais de alocação de recursos. Com base nos conceitos apresentados no artigo, foram implementadas duas soluções exatas para o problema clássico — Backtracking e Programação Dinâmica — e aplicada a modelagem a um cenário de planejamento de missões espaciais. Os resultados experimentais indicam que ambas as abordagens alcançam a solução ótima, com a Programação Dinâmica apresentando melhor desempenho para instâncias maiores e o Backtracking sendo mais adequado para instâncias pequenas.*

Abstract. *This work presents the analysis of the article The Price-Elastic Knapsack Problem (Fukasawa et al., 2024), published in an IEEE-indexed journal, which addresses a variation of the 0/1 Knapsack Problem by incorporating price elasticity, making it closer to real-world resource allocation scenarios. Based on the concepts introduced in the article, two exact solutions for the classical problem — Backtracking and Dynamic Programming — were implemented, and the modeling was applied to a space mission planning scenario. Experimental results indicate that both approaches achieve the optimal solution, with Dynamic Programming showing better performance for larger instances, while Backtracking is more suitable for small instances.*

1. Introdução

O Problema da Mochila 0/1 é um problema clássico de otimização combinatória pertencente à classe NP-completo. Ele consiste em selecionar um subconjunto de n itens, cada um com um peso p_i e um valor v_i, de forma a maximizar o valor total sem exceder a capacidade C da mochila. A formulação 0/1 indica que cada item pode ser escolhido integralmente ou não ser escolhido, não sendo permitida a fração de itens.

A formulação matemática é dada por:

$$\text{Maximizar:} \quad \sum_{i=1}^{n} v_i x_i$$

$$\text{Sujeito a:} \quad \sum_{i=1}^{n} p_i x_i \leq C, \quad x_i \in \{0,1\}$$

Figura 1 – Fórmula matemática do problema da mochila 0/1 clássico

Esse problema possui ampla aplicação prática, como em logística, seleção de investimentos, alocação de recursos e planejamento de missões. Sua relevância é ainda maior em cenários com restrições severas de peso e capacidade, como missões espaciais.

2. Fundamentação Teórica

2.1 O Problema da Mochila e sua Relevância

A mochila 0/1 tem sido estudada extensivamente na ciência da computação e pesquisa operacional, sendo base para problemas mais complexos, como a mochila multidimensional e a mochila com restrições temporais. Sua importância prática está relacionada à necessidade de otimização em ambientes de recursos limitados, onde a seleção correta de itens é crucial para o sucesso da operação.

2.2 Análise do Artigo "The Price-Elastic Knapsack Problem"

O trabalho de Fukasawa et al. (2024) apresenta uma extensão do problema clássico incorporando elasticidade de preços, em que o valor dos itens varia conforme a quantidade selecionada. Essa abordagem aproxima o modelo de situações reais, como compras em grande escala, descontos progressivos e variação de demanda.

Os autores propõem:

- Formulação matemática não linear para capturar a elasticidade.
- Métodos exatos usando programação inteira mista.
- Estratégias de relaxação e branch-and-bound para acelerar a resolução.

- Testes computacionais mostrando ganhos de desempenho em instâncias com alta elasticidade.

3. Justificativa

Problemas de otimização com restrições rígidas são comuns em operações críticas, como missões espaciais. A inclusão de elasticidade de preços, como feito no artigo base, aproxima o modelo de situações reais, permitindo decisões mais precisas e economicamente viáveis.

4. Metodologia

Para modelar e resolver o Problema da Mochila com Elasticidade de Preço, os autores começam definindo uma formulação matemática capaz de capturar a variação do valor de cada item em função da quantidade adquirida. Essa relação não linear entre preço, quantidade e valor total é incorporada diretamente na função objetivo, tornando o modelo mais realista para aplicações econômicas e logísticas.

4.1 Formulação Matemática

O problema é formulado como um modelo de Programação Inteira Mista (MIP) com as seguintes definições:

- **Variáveis de decisão**

$$x_i \in \mathbb{Z}^+ \quad \text{ou} \quad x_i \in \{0,1\}$$

Figura 2 – Variáveis de decisão do modelo

onde x_i representa a quantidade (ou presença) do item i na solução.

- **Função objetivo**

$$\max \sum_{i=1}^{n} p_i(x_i) \cdot x_i$$

Figura 3 – Função objetivo do problema

onde $p_i(x_i)$ representa o preço ou valor unitário do item i, dependente de x_i, modelando a elasticidade de preço.

- **Restrição de capacidade**

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

Figura 4 – Restrição de capacidade

onde w_i é o peso (ou volume) do item i e C é a capacidade máxima da mochila.

- **Outras restrições**

Limites de demanda, disponibilidade e viabilidade técnica, conforme o contexto da aplicação.

Como essa formulação é não linear devido a $p_i(x_i)$, os autores aplicam técnicas de linearização para reescrever a função objetivo em termos lineares por partes, permitindo que o problema seja tratado com algoritmos de programação inteira.

4.2 Estratégia de Resolução

O processo de resolução é baseado no método Branch-and-Bound, explorando sistematicamente o espaço de soluções e usando limites superiores e inferiores para podar ramos inviáveis.

Para melhorar a eficiência, são empregadas estratégias de pré-processamento, como:

- Eliminação de variáveis irrelevantes.
- Redução de intervalos de busca.
- Agrupamento de itens com características semelhantes.

4.3 Testes Computacionais

Os autores realizam experimentos em instâncias com diferentes níveis de elasticidade e capacidade, comparando:

- O desempenho do modelo proposto.
- Os resultados obtidos pelo modelo clássico sem elasticidade.

A metodologia também inclui análises de sensibilidade, avaliando como alterações nos parâmetros de elasticidade afetam a qualidade da solução e o tempo computacional.

5. Implementação das Soluções

5.1 Backtracking

O algoritmo de Backtracking implementa uma busca em profundidade (DFS — *Depth First Search*) para explorar todas as combinações possíveis de itens. A cada nível da recursão, são consideradas duas possibilidades para o item atual:

1. **Não incluir o item na solução parcial.**
2. **Incluir o item, desde que a capacidade da mochila não seja excedida.**

A técnica de poda é utilizada para interromper a busca em ramos inviáveis, ou seja, quando o peso acumulado ultrapassa a capacidade C. Ao atingir o último item, o algoritmo compara o valor total obtido com o melhor valor encontrado até então e, se for maior, atualiza a solução ótima.

Algoritmo 1: Mochila_Backtracking

Entrada: peso[1..n], valor[1..n], C

Saída: MelhorValor, MelhorEscolha[1..n]

1 MelhorValor ← 0

2 MelhorPeso ← 0

3 EscolhaAtual[1..n] ← 0

4 MelhorEscolha[1..n] ← 0

```
5 procedimento DFS(idx, pesoAtual, valorAtual):
6     se pesoAtual > C então
7         retornar
8     fim-se
9     se idx > n então
10        se valorAtual > MelhorValor então
11            MelhorValor ← valorAtual
12            MelhorPeso  ← pesoAtual
13            MelhorEscolha ← cópia(EscolhaAtual)
14        fim-se
15        retornar
16    fim-se
17
18    EscolhaAtual[idx] ← 0
19    DFS (idx + 1, pesoAtual, valorAtual)
20
21    EscolhaAtual[idx] ← 1
22    DFS (idx + 1, pesoAtual + peso[idx], valorAtual + valor[idx])
23
24 ler n, C e os vetores peso, valor
25 DFS (1, 0, 0)
26 retornar (MelhorValor, MelhorEscolha)
```

Apesar de garantir a solução exata, o custo computacional no pior caso é $O(2^n)$, sendo adequado apenas para instâncias pequenas ou médias. Esse método é importante como referência para comparação com algoritmos mais eficientes.

5.2 Programação Dinâmica

O algoritmo de Programação Dinâmica bidimensional constrói uma matriz dp[i][c], onde cada elemento representa o melhor valor possível utilizando apenas os i primeiros itens com capacidade máxima c.

A construção da tabela segue a lógica:

- **Não pegar o item i:** o valor permanece o mesmo que para dp [i − 1] [c].
- **Pegar o item i:** o valor é calculado como o valor de dp [i − 1] [c − peso[i]] somado ao valor do item i, desde que peso[i] ≤ c.

Ao final, o valor ótimo é encontrado em dp[n][C] e o vetor de escolha dos itens é reconstruído percorrendo a tabela de trás para frente, verificando quais itens foram incluídos na solução final. Este método possui complexidade O(n·C), sendo mais eficiente para instâncias maiores quando comparado ao Backtracking, especialmente se a capacidade C não for muito grande.

Algoritmo 2: Mochila_DP_2D

Entrada: peso[1..n], valor[1..n], C

Saída: ValorÓtimo, Escolha[1..n]

```
1 criar matriz dp [0..n][0..C] inicializada com 0


2 para i ← 1 até n faça
3     para c ← 0 até C faça
4         naoPega ← dp[i-1] [c]
5         se peso[i] ≤ c então
6             pegar ← dp[i-1] [c - peso[i]] + valor[i]
7         senão
8             pegar ← -∞
```

```
9         fim-se
10        dp[i][c] ← max(naoPega, pega)
11    fim-para
12 fim-para


13 ValorÓtimo ← dp[n][C]
14 Escolha [1..n] ← 0
15 c ← C
16 para i ← n até 1 passo -1 faça
17    se dp[i][c] ≠ dp[i-1] [c] então
18        Escolha[i] ← 1
19        c ← c - peso[i]
20    fim-se
21 fim-para
22 retornar (ValorÓtimo, Escolha)
```

6. Aplicação: Planejamento de Missão Espacial

6.1 Situação

Uma equipe de cientistas planeja enviar um rover para um novo planeta a fim de coletar amostras e executar experimentos. O veículo possui capacidade limitada de carga (massa) e há uma lista de equipamentos/suprimentos possíveis. Cada item tem um peso (massa) e uma utilidade/valor que mede sua contribuição científica e operacional para a missão.

6.2 Problema

Selecionar o subconjunto de itens que maximize a utilidade total da missão sem exceder a capacidade total de massa do rover.

6.3 Dados da instância

A Tabela 1 resume os itens candidatos ao payload do rover, com suas respectivas massas e utilidades (valores), usados na instância de 100 kg.

Tabela 1 – Itens candidatos da missão espacial (instância C = 100 kg)

Item	Massa (kg)	Valor
Câmera de alta resolução	20	40
Braço Robótico	50	100
Analisador de solo	30	60
Detector de radiação	10	30
Fonte de energia extra	40	70

6.4 Algoritmo A — Backtracking (seleção de payload)

Implementa uma busca em profundidade (DFS) no espaço de soluções. Para cada item i, o algoritmo considera não incluir e incluir o item, podando ramos cujo peso acumulado ultrapasse C. Ao atingir o último item, compara o valor obtido com o melhor valor conhecido e atualiza a solução ótima (valor total, massa total e lista de itens). É exato, porém com custo exponencial $O(2^n)$; indicado para instâncias pequenas/médias e como referência para validar outras abordagens.

Pseudocódigo (Algoritmo — Missão_Backtracking).

Entrada: itens[1..n] com (massa[i], util[i], nome[i]), capacidade C
Saída: MelhorValor, MelhorMassa, MelhorEscolha[1..n]

```
1  MelhorValor ← 0; MelhorMassa ← 0
2  EscolhaAtual[1..n] ← 0; MelhorEscolha[1..n] ← 0

3  procedimento DFS(i, massaAtual, valorAtual):
4     se massaAtual > C então retornar
5     se i > n então
6        se valorAtual > MelhorValor então
7           MelhorValor ← valorAtual
8           MelhorMassa ← massaAtual
9           MelhorEscolha ← cópia(EscolhaAtual)
10       fim-se
11       retornar
12    fim-se
13
14    EscolhaAtual[i] ← 0
15    DFS(i+1, massaAtual, valorAtual)
```

```
16
17    EscolhaAtual[i] ← 1
18    DFS(i+1, massaAtual + massa[i], valorAtual + util[i])
19
20 DFS(1, 0, 0)
21 retornar (MelhorValor, MelhorMassa, MelhorEscolha)
```

6.5 Algoritmo B — Programação Dinâmica 2D (seleção de payload)

Constrói-se uma matriz dp[i][c] que armazena a melhor utilidade ao considerar os i primeiros itens e uma capacidade disponível c. Para cada estado:

- **Não pegar** o item i: dp[i−1][c].
- **Pegar** o item i: $dp[i-1][c-m_i]+u_i$, se $m_i \leq c$.
  Ao final, dp[n][C] contém o **valor ótimo**. A **reconstrução** da solução percorre a tabela de trás para frente para descobrir quais itens foram escolhidos. Complexidade O(n·C) em tempo e memória.

Pseudocódigo (Algoritmo — Missão_DP_2D).

Entrada: itens[1..n] com (massa[i], util[i], nome[i]), capacidade C
Saída: ValorÓtimo, Escolha[1..n]

```
1  criar dp[0..n][0..C] ← 0
2  para i ← 1..n:
3     para c ← 0..C:
4        naoPega ← dp[i-1][c]
5        se massa[i] ≤ c então
6           pega ← dp[i-1][c - massa[i]] + util[i]
7        senão
8           pega ← naoPega
9        fim-se
10       dp[i][c] ← max(naoPega, pega)

11 ValorÓtimo ← dp[n][C]
12 Escolha[1..n] ← 0; c ← C
13 para i ← n..1 passo -1:
14    se dp[i][c] ≠ dp[i-1][c] então
15       Escolha[i] ← 1
16       c ← c - massa[i]
17    fim-se
18 retornar (ValorÓtimo, Escolha)
```

6.6 Demonstração matemática das recorrências e complexidades

Recorrência ótima do problema. Defina **OPT(i,c)** como o melhor valor que pode ser obtido usando apenas os i primeiros itens com capacidade disponível **c**. Então,

$$\mathrm{OPT}(i,c) = \begin{cases} 0, & \text{se } i = 0 \text{ ou } c = 0, \\ \max\{\mathrm{OPT}(i-1,c),\ \mathrm{OPT}(i-1,c-m_i)+u_i\}, & \text{se } m_i \le c, \\ \mathrm{OPT}(i-1,c), & \text{se } m_i > c. \end{cases}$$

Figura 5 - Equação de recorrência ótima do problema da mochila

A corretude decorre do princípio da optimalidade: para o item **i** viável **($m_i \le c$)**, a solução ótima ou não o utiliza, ou o utiliza e soma **u_i** ao ótimo residual com capacidade **$c-m_i$**; se **$m_i > c$**, o item é inviável.

Backtracking: recorrência de custo e ordem de tempo. Seja **T(i,c)** o tempo para explorar a subárvore a partir do item **i** com capacidade **c**. Quando **$m_i \le c$**,

$$T(i,c) \le T(i+1,c) + T(i+1,c-mi) + O(1),$$

e, quando **$m_i > c$**, **$T(i,c) \le T(i+1,c)+O(1)$**. No pior caso (poda pouco efetiva), essa exploração se comporta como uma árvore binária completa de profundidade **n**, implicando

$$T(1)=\Theta(2^n).$$

O uso de memória é determinado pela profundidade da recursão e vetores auxiliares, logo **$S(n)=\Theta(n)$**.

Programação Dinâmica 2D: custo e ordem de tempo. A DP computa **dp[i][c] = OPT(i,c)** para todos os **(i,c)**. Cada estado é obtido em **O(1)** a partir da recorrência ótima; como existem **(n+1) (C+1)** estados, o custo total é

$$T(n,C)=\Theta(nC).$$

O espaço para a tabela 2D é **$S(n, C) = \Theta(nC)$**; pode-se reduzir para **$\Theta(C)$** com DP 1D (atualização de **c** decrescente), à custa de cuidados na reconstrução da solução. Trata-se de um tempo pseudo-polinomial (linear em **n** e em **C**, não em **logC**).

Em termos práticos, a escolha depende de comparar **2^n** com **nC**. O backtracking é viável quando n é pequeno a ponto de **$2^n \lesssim nC$** (por exemplo, com **$C=10^3$**, o cruzamento ocorre por volta de **$n \approx 13$–14**); acima disso, a DP tende a ser superior por oferecer custo

previsível e reconstrução direta da solução. Para capacidades muito grandes, preferem-se otimizações da DP (1D, reescalonamento) ou esquemas aproximados.

7. Conclusão

Este trabalho analisou o artigo *The Price-Elastic Knapsack Problem* (Fukasawa, Naoum-Sawaya e Oliveira, 2024), que amplia o modelo clássico da mochila ao incorporar elasticidade de preço/valor, aproximando a modelagem de cenários reais de alocação de recursos. A partir dessa base conceitual, implementamos duas abordagens exatas para o problema 0/1 — Backtracking (busca em profundidade com poda) e Programação Dinâmica (tabela 2D com reconstrução da solução).
No estudo de caso do planejamento de missão espacial, ambas as abordagens encontraram a solução ótima para a instância proposta (capacidade de 100 kg), evidenciando que:

1. O Backtracking é útil como referência e para instâncias pequenas, por garantir otimalidade e permitir auditoria da solução;
2. A Programação Dinâmica apresenta melhor desempenho para capacidades moderadas, com tempo previsível e reconstrução direta do payload ótimo.

Referências

1. **FUKASAWA, R.; NAOUM-SAWAYA, J.; OLIVEIRA, D.** The Price-Elastic Knapsack Problem. *Omega*, v. 124, 2024, art. 103003. DOI: **10.1016/j.omega.2023.103003**.
2. **IZIDORIO, E. H. de A.; LEAL, S. da C.** *FinalProject_DCC606_Problema_da_Mochila_8_RR_2025*. GitHub, 2025. Disponível em: **https://github.com/EhoKira/FinalProject_DCC606_Problema_da_Mochila_8_RR_2025.git**. (Acesso em: 13 ago. 2025.)