Análise Exaustiva e Implementação de Alta Performance para o Problema da Mochila: Da Escolha Gulosa à Programação Dinâmica

Caio Wallace Machado Gomes[1], **Eduardo Fraga Pereira**[1], **Murilo Honorato de Souza**[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Campus Samambaia – Caixa Postal 131 – 74.001-970 – Goiânia – GO – Brasil

{caio.machado, eduardofraga, murilo.honorato}@discente.ufg.br

Abstract. *This report provides a comprehensive study and evaluation of algorithms for the Knapsack Problem, exploring both Fractional and 0/1 variants. We detail an high-performance implementation in C++, employing a class-based architecture to solve instances with over* 10^7 *items. The work analyzes the Greedy Choice Property for the fractional case and explores the optimal substructure through Dynamic Programming for the 0/1 case. Extensive benchmarks are provided, comparing recursive approaches with optimized DP (1D space optimization). Furthermore, we describe a complete automation pipeline involving Python-based data generation and Makefile orchestration for execution and performance profiling.*

Resumo. *Este relatório provê um estudo exaustivo e avaliação de algoritmos para o Problema da Mochila, explorando as variantes Fracionária e 0/1. Detalhamos uma implementação de alta performance em C++, empregando uma arquitetura baseada em classes para resolver instâncias com mais de* 10^7 *itens. O trabalho analisa a Propriedade da Escolha Gulosa para o caso fracionário e explora a subestrutura ótima via Programação Dinâmica para o caso 0/1. São apresentados benchmarks extensos, comparando abordagens recursivas com DP otimizada (espaço 1D). Adicionalmente, descrevemos um pipeline de automação completo envolvendo geração de dados em Python e orquestração via Makefile para execução e perfilamento de desempenho.*

1. Introdução

O Problema da Mochila (*Knapsack Problem*) é um dos modelos mais fundamentais na teoria de algoritmos e otimização. Sua formulação básica — selecionar um conjunto de itens sob uma restrição de peso para maximizar o valor — serve como abstração para inúmeros problemas do mundo real, desde a otimização de orçamentos publicitários até o empacotamento de dados em redes de comunicação.

A importância deste problema reside não apenas em sua aplicação direta, mas no que ele nos ensina sobre complexidade computacional. Enquanto o problema da mochila fracionária possui uma solução gulosa eficiente, a versão 0/1 é um problema NP-difícil, servindo de porta de entrada para o estudo de Programação Dinâmica e algoritmos de aproximação [CP-Algorithms 2024]. Este projeto visa implementar ambas as soluções utilizando C++, focando em eficiência assintótica e prática, além de documentar o processo de desenvolvimento e validação técnica em cenários de maratona de programação [Guide 2024].

2. Modelagem Matemática e Teórica

Para fins de rigor acadêmico, definimos o problema formalmente. Seja N o número de itens disponíveis. Cada item i possui um valor $v_i > 0$ e um peso $p_i > 0$. A mochila possui uma capacidade máxima W.

2.1. A Mochila Fracionária

O objetivo é encontrar frações x_i ($0 \leq x_i \leq 1$) para cada item i que maximizem a função objetivo:

$$V = \sum_{i=1}^{N} v_i x_i$$

sujeito à restrição de capacidade:

$$\sum_{i=1}^{N} p_i x_i \leq W$$

Nesta variante, a **Propriedade da Escolha Gulosa** é válida: uma escolha local ótima (o item com maior densidade de valor v_i/p_i) faz parte de uma solução global ótima, conforme detalhado em guias de algoritmos [GeeksforGeeks 2024, (TakeU-Forward) 2023].

2.2. A Mochila 0/1

Nesta variante, a restrição torna-se binária: $x_i \in \{0, 1\}$. Aqui, o problema deixa de ser linear e passa a ter caráter combinatório. A estrutura do problema apresenta **Subestrutura Ótima**, permitindo que a solução para uma capacidade W seja construída a partir da recorrência:

$$DP[i][w] = \max(DP[i-1][w], v_i + DP[i-1][w - p_i])$$

3. Arquitetura e Implementação

3.1. Justificativa da Linguagem C++

A escolha do **C++** como linguagem principal fundamentou-se em sua eficiência de baixo nível. Para processar os arquivos de teste gerados, que chegam a 10^7 itens (aproximadamente 120MB de texto), o C++ oferece vantagens cruciais:

- **Gestão de I/O**: Uso de fluxos (*fast I/O*) para leitura rápida de dados.
- **Controle de Memória**: O uso de `std::vector::reserve` permitiu alocar o espaço exato para os itens, evitando múltiplas realocações durante o *parsing*.
- **Performance**: O compilador GCC, com a flag de otimização `-O3`, gera código de máquina altamente eficiente para loops de processamento e ordenação.

3.2. Implementação Orientada a Objetos

A implementação foi estruturada utilizando os princípios de **Orientação a Objetos (POO)**. A criação da classe `Knapsack` trouxe diversos benefícios:

1. **Encapsulamento**: O estado do problema (itens e capacidade) é protegido e gerido internamente.
2. **Modularidade**: Permite a coexistência de múltiplos algoritmos (Guloso, Recursivo, DP) operando sobre o mesmo conjunto de dados de forma isolada.
3. **Manutenibilidade**: A separação clara entre a interface (`.hpp`) e a implementação (`.cpp`) facilita a expansão futura do projeto para testes com memoização ou outros paradigmas.

4. Metodologia Experimental e Automação

4.1. Geração de Casos de Teste (Python)

Embora não faça parte do pacote final de entrega, foi utilizado um script auxiliar em Python para a geração sistemática de cargas de trabalho sintéticas. O objetivo foi validar a corretude e a escalabilidade dos algoritmos em diferentes proporções. Os parâmetros fundamentais utilizados na geração foram:

- **Densidade**: Itens com pesos p_i variando entre $[1, 5 \times 10^5]$ e valores v_i entre $[10, 10^6]$.
- **Proporcionalidade**: A capacidade da mochila W foi ajustada dinamicamente, mantendo uma proporção média de $0,5 \times N$ a $5 \times N$, garantindo que o espaço da mochila fosse um limitador real para o preenchimento guloso e um desafio de memória para a Programação Dinâmica.

4.2. Orquestração via Makefile

O processo de build e teste foi automatizado com um **Makefile**. Além de gerir a compilação com as flags `-O3` e `-Wall`, o script implementa regras dinâmicas que automatizam a execução contra a base de dados em `dados/`, facilitando o perfilamento de tempo em diferentes escalas de N.

5. Análise de Algoritmos e Complexidade

5.1. Algoritmo Guloso

O algoritmo para a mochila fracionária possui complexidade de tempo dominada pela ordenação: $O(N \log N)$. O loop de preenchimento é linear $O(N)$. Space-wise, utiliza $O(N)$ para armazenar o vetor de itens.

Para a mochila 0/1, implementamos a técnica de **1D Dynamic Programming**. Em vez de uma matriz $N \times W$, utilizamos apenas um vetor `dp[W+1]`. Isso reduz drasticamente o consumo de memória, permitindo resolver casos onde W é grande sem estourar a RAM do sistema. A complexidade de tempo permanece $O(NW)$.

6. Pseudocódigos

Nesta seção, apresentamos as representações em alto nível dos algoritmos implementados.

Listing 1. Mochila Fracionária (Guloso)

```
Procedure Fracionaria(itens, W)
    Ordenar itens por razao v/p decrescente
    valorTotal <- 0
    For cada item i em itens
        If p_i <= W
            valorTotal <- valorTotal + v_i
            W <- W - p_i
        Else
            fracao <- W / p_i
            valorTotal <- valorTotal + (v_i * fracao)
            parar
        End If
    End For
    Return valorTotal
End Procedure
```

Listing 2. Mochila 0/1 (Programação Dinâmica 1D)

```
Procedure BinariaDP(itens, W)
    Criar vetor DP de tamanho W+1 inicializado com 0
    For cada item i em itens
        For w de W ate p_i
            DP[w] <- max(DP[w], v_i + DP[w-p_i])
        End For
    End For
    Return DP[W]
End Procedure
```

7. Resultados e Discussão

7.1. Complexidade e Demonstrativo de Desempenho

A análise experimental ratificou as previsões teóricas. A Tabela 1 apresenta os dados coletados em ambiente controlado. Observamos que o tempo de execução da Mochila Fracionária cresce de forma loglinear ($N \log N$), o que é imperceptível para o usuário até que N atinja a casa dos milhões.

Já para a Mochila 0/1, a diferença entre a abordagem recursiva e a de Programação Dinâmica é abismal. Na recursão pura, a cada novo item adicionado, o número de chamadas de função dobra, levando ao estouro do tempo de execução para instâncias pequenas (acima de 45 itens). A Programação Dinâmica, ao utilizar a técnica de memorização (neste caso, por meio de um vetor), transforma esse custo em um tempo proporcional ao produto do número de itens pela capacidade ($O(nW)$), conceito central em treinamento para competições [Laaksonen 2018].

Tabela 1. Benchmarks de Execução Completos (Tempo de CPU)

N (Itens)	Capacidade	Guloso	Rec. (Binário)	DP (Binário)
10	50	< 0.0001s	0.0000s	< 0.0001s
20	100	< 0.0001s	0.0001s	< 0.0001s
30	150	< 0.0001s	0.0065s	< 0.0001s
40	200	< 0.0001s	2.5894s	< 0.0001s
50	250	< 0.0001s	44 min*	< 0.0001s
60	300	< 0.0001s	31 dias*	< 0.0001s
70	350	< 0.0001s	88 anos*	< 0.0001s
80	400	< 0.0001s	90 mil anos*	< 0.0001s
90	450	< 0.0001s	92 milh. anos*	< 0.0001s
100	500	< 0.0001s	94 bilh. anos*	< 0.0001s
1.000	2.000	0.0001s	-	0.0010s
10.000	10.000	0.0005s	-	0.0317s
100.000	50.000	0.0061s	-	1.5853s
1.000.000	500.000	0.0742s	-	147.60s
10.000.000	5.000.000	0.8935s	-	4.1 h*

* Valores estimados baseados na complexidade assintótica do algoritmo e performance prévia.

7.2. Análise de Memória e Otimização 1D

Um dos maiores desafios técnicos enfrentados foi a gestão de memória para a Mochila 0/1. Em uma abordagem tradicional com matriz bidimensional $DP[N][W]$, o consumo de memória para uma instância de 10^4 itens e capacidade 10^6 ultrapassaria facilmente os 40GB (considerando valores *double* de 8 bytes), tornando a execução impossível em hardware convencional.

Como observado em [CP-Algorithms 2024], a otimização para vetor unidimensional é possível pois o estado atual f_i depende exclusivamente do estado imediatamente anterior f_{i-1}. Ao removermos a primeira dimensão da matriz, obtemos uma regra de transição que deve ser executada obrigatoriamente em ordem decrescente de capacidade j. Essa ordem garante que o valor f_{j-p_i} ainda corresponda ao estado do passo $i-1$ (não tendo sido sobrescrito pelo passo i), mantendo o invariante necessário para o problema da mochila 0/1. Esta implementação reduziu drasticamente o requisito de memória para $O(W)$, permitindo que o sistema processasse as mesmas instâncias utilizando apenas alguns megabytes de RAM.

8. Uso de Ferramentas Inteligentes

A integração de ferramentas de IA (Antigravity) foi fundamental para a agilidade do projeto. Seu uso distribuiu-se em:

- **Automação de Infraestrutura e Visualização**: Criação do script de geração de testes, estruturação do `Makefile` e desenvolvimento de scripts Python para a geração de gráficos comparativos de performance.

- **Revisão Técnica**: O assistente foi utilizado para realizar a revisão do código-fonte C++, sugerindo melhorias de legibilidade e padrões de documentação.
- **Revisão de Documentação**: Apoio na correção ortográfica e estruturação gramatical deste relatório técnico.

9. Considerações Finais

O trabalho demonstrou que a eficiência em algoritmos de busca e otimização não depende apenas da capacidade do processador, mas da escolha adequada do paradigma (Guloso vs. Dinâmico) e de uma implementação que respeite os limites de hardware da linguagem escolhida.

Referências

CP-Algorithms (2024). Knapsack problem - dynamic programming. `https://cp-algorithms.com/dynamic_programming/knapsack.html`.

GeeksforGeeks (2024). Fractional knapsack problem. `https://www.geeksforgeeks.org/fractional-knapsack-problem/`.

Guide, U. (2024). Knapsack - introduction to dynamic programming. `https://usaco.guide/gold/knapsack/`.

Laaksonen, A. (2018). *Competitive Programmer's Handbook*. Codeforces.

(TakeUForward), S. (2023). Fractional knapsack problem - greedy algorithm. `https://takeuforward.org/data-structure/fractional-knapsack-problem-greedy-approach/`.