Relatorio do Problema da Mochila 0/1 de Introdução à Ciência da Computação II

Leonardo Kenzo Tanaka e Pedro Teidi de Sá Yamacita

8 de outubro de 2025

1 Introdução

O Problema da Mochila 0/1 é um problema clássico de otimização combinatória onde, dado um conjunto de n itens (cada um com peso w_i e valor v_i) e uma mochila com capacidade máxima W, deve-se determinar a combinação de itens que maximiza o valor total sem exceder a capacidade da mochila. Este relatório apresenta três abordagens para resolver o problema:

- Força Bruta: Explora todas as combinações possíveis
- Algoritmo Guloso: Seleciona itens com maior razão valor/peso
- Programação Dinâmica: Utiliza subestrutura ótima e memorização

2 Implementações e Análise do código

2.1 Estruturas de Dados

```c
struct item_{
    float peso;
    float valor;
    int chave;
};

struct mochila_{
    NO *inicio;
    NO *fim;
    int quantidadeItens;
    float pesoAtual;
    float pesoMaximo;
    float valorTotal;
};

typedef struct no_{
    ITEM *item;
    struct no_ *proximo;
    struct no_ *anterior;
};
```

Foi utilizada uma lista duplamente encadeada para armazenar os itens selecionados e controlar o peso e o valor totais, estrutura empregada apenas nos algoritmos de força bruta e guloso. Cada item possui seu respectivo peso, valor e uma identificação única, definidos conforme a ordem de entrada dos dados (input).

2.2 Algoritmo de Força Bruta

Gera todas as 2^n combinações possíveis de itens usando manipulação de bits e seleciona a combinação com maior valor que respeita a restrição de capacidade.

Implementação:

```c
void ForcaBruta(ITEM **todosItens, int quantItens, int pesoMaximo){
    //Limita a quantidade de itens para 25, se for maior que isso fica muito longo
    if(todosItens && quantItens <= 25){
        MOCHILA *mochila = MochilaCriar(pesoMaximo);
        int melhorValor = 0, melhorPeso = 0;
        int melhorCombinacao;

        //2^quantItens Ex: 16 combinacoes = 10000 binario
        int totalCombinacoes = 1 << quantItens;

        //Testa todas as combinacoes possiveis
        for(int combinacoes = 1; combinacoes < totalCombinacoes; combinacoes++){
            //Garante que a mochila esta vazia
            MochilaEsvaziar(mochila);
            bool combinacaoValida = true;

            //Teste de combinacoes de itens
            for(int i = 0; i < quantItens; i++){
            //Seleciona o item a ser adicionado na mochila atraves de operacao com bits
                if(combinacoes & (1 << i)){
                    if(MochilaCabe(mochila, todosItens[i])){
                        MochilaAdicionarItem(mochila, todosItens[i]);
                    }
                    else{
                        combinacaoValida = false;
                        break;
                    }
                }
            }
            //Escolhe a melhor combinacao
            if(combinacaoValida && MochilaGetValor(mochila) >= melhorValor){
                melhorValor = MochilaGetValor(mochila);
                melhorCombinacao = combinacoes;
            }
        }
        //Calcula o melhor peso
        for(int i = 0; i < quantItens; i++){
            if(melhorCombinacao & (1 << i))
                melhorPeso += ItemGetPeso(todosItens[i]);
        }
    }
}
```

O uso da força bruta se torna inviável para valores de n maiores que 50, pois sua complexidade é $O(2^n)$, crescendo de forma exponencial. Além disso, o algoritmo utiliza operações bit a bit para gerar todas as combinações possíveis de itens na mochila.

Por exemplo, com 4 itens, são geradas 16 combinações. Isso pode ser representado por números binários de 4 bits (de 0000 a 1111), em que cada bit corresponde a um item — o valor 1 indica que o item foi incluído na mochila, enquanto 0 indica que não foi.

Análise:

- Loop externo: 2^n - 1 iterações (todas as combinações não-vazias)
- Operações de verificação: O(1) cada

Complexidade:

- $T(n) = O(2^n)$

2.3 Algoritmo Guloso

Ordena os itens pela razão valor/peso (decrescente) e seleciona itens sequencialmente até a capacidade ser atingida, foi utilizado o Bubble Sort para ordenar a lista de itens, então sua complexidade será no mínimo $O(n^2)$ pela implementação utilizada.
Além disso, o algoritmo guloso não garante a solução ótima, somente uma aproximação da solução, já que ele considera somente as razões valor/peso.
Implementação:

```c
void Guloso(ITEM **todosItens, int quantItens, int pesoMaximo){
    if(!todosItens || quantItens <= 0){
        return;
    }

    ITEM **ordenados = (ITEM **)calloc(quantItens, sizeof(ITEM *));
    bool *usado = (bool *)calloc(quantItens, sizeof(bool));
    for(int i = 0; i< quantItens; i++){
        ordenados[i] = todosItens[i];
        usado[i] = false;
    }
    //Ordena os itens pela razao valor/peso usando Bubble Sort
    for(int i = 0; i < quantItens; i++){
        int melhor = -1;
        int melhorRazao = -1;
        for(int j = 0; j < quantItens; j++){
            if(!usado[j] && todosItens[j] != NULL){
                int razao = ItemGetRazao(todosItens[j]);
                if(razao > melhorRazao){
                    melhorRazao = razao;
                    melhor = j;
                }
            }
        }
        if (melhor != -1) {
            ordenados[i] = todosItens[melhor];
            usado[melhor] = true;
        }
    }
    MOCHILA *mochila = MochilaCriar(pesoMaximo);
    int melhorValor = 0, melhorPeso = 0;
    int *itensSelecionados = (int *)calloc(quantItens, sizeof(int));
    int quantidadeSelecionados = 0;

    //Seleciona os itens com a melhor razao
    for(int i = 0; i < quantItens; i++){
        if(ordenados[i] && MochilaCabe(mochila, ordenados[i])){
            MochilaAdicionarItem(mochila, ordenados[i]);
            itensSelecionados[quantidadeSelecionados++] = ItemGetChave(ordenados[i]);
            melhorPeso += ItemGetPeso(ordenados[i]);
            melhorValor += ItemGetValor(ordenados[i]);
        }
    }
}
```

O algoritmo guloso falha em entregar a solução ótima do problema da mochila 0/1 porque ao selecionar o item com melhor razão pode bloquear combinações melhores e a solução ótima não necessariamente contém o item de melhor razão

Análise:

- Ordenação por bubble sort: O(n²)
- Loop único: n iterações

Complexidade¿

- $T(n) = O(n^2)$

2.4 Programação Dinâmica

O algoritmo constrói uma tabela DP[i][w], que armazena o valor máximo obtido utilizando os i primeiros itens e uma capacidade w. Esse método aplica a estratégia de dividir o problema em subproblemas menores, resolvendo cada um deles de forma independente e, em seguida, combinando as soluções parciais para obter a resposta do problema completo.

Implementação:

```c
void ProgramacaoDinamica(ITEM **todosItens, int quantItens, int pesoMaximo){
    if(!todosItens){
        return;
    }
    //Aloca memoria
    int **programacaoDinamica = (int **)calloc(quantItens + 1, sizeof(int *));
    for(int i = 0; i <= quantItens; i++){
        programacaoDinamica[i] = (int *)calloc(pesoMaximo + 1, sizeof(int));
    }
    //Itera pelos itens e pelos pesos de cada item
    for(int i = 1; i <= quantItens; i++){
        for(int p = 0; p <= pesoMaximo; p++){
            //Nao pega o item
            programacaoDinamica[i][p] = programacaoDinamica[i - 1][p];

            //Pega o item
            if (ItemGetPeso(todosItens[i - 1]) <= p) {
                int valorComItem = programacaoDinamica[i - 1][p -
                    (int)ItemGetPeso(todosItens[i - 1])] + ItemGetValor(todosItens[i -
                    1]);
                if (valorComItem > programacaoDinamica[i][p]) {
                    programacaoDinamica[i][p] = valorComItem;
                }
            }
        }
    }
    //Rastreia quais itens foram selecionados
    int p = pesoMaximo;
    int *itensSelecionados = (int *)calloc(quantItens, sizeof(int));
    int quantidadeSelecionados = 0;
    int melhorValor = 0;
    int melhorPeso = 0;
    for (int i = quantItens; i > 0 && p > 0; i--) {

        //Se o valor mudou, o item i-1 foi incluido
        if (programacaoDinamica[i][p] != programacaoDinamica[i - 1][p]) {
            itensSelecionados[quantidadeSelecionados] = i - 1;
            quantidadeSelecionados++;
            melhorPeso += ItemGetPeso(todosItens[i - 1]);
            p -= ItemGetPeso(todosItens[i - 1]);
            melhorValor += ItemGetValor(todosItens[i - 1]);
        }
    }
    //Inverter ordem dos itens (foram adicionados de tras para frente)
    for (int i = 0; i < quantidadeSelecionados / 2; i++) {
        int temp = itensSelecionados[i];
        int j = quantidadeSelecionados - 1 - i;
        itensSelecionados[i] = itensSelecionados[j];
        itensSelecionados[j] = temp;
    }
}
```

Análise:

- Loop externo (itens): n iterações
- Loop interno (capacidades): W iterações
- Operações por célula: O(1)

Complexidade:

- $T(n, W) = O(n \times W)$

Equação de recorrência:

- Seja DP[i][w] = valor máximo obtível com os primeiros i itens e capacidade w.

Condições Base:

- DP[0][w] = 0, para todo $w \in [0, W]$
- DP[i][0] = 0, para todo $i \in [0, n]$

Equação de Recorrência:
Para $i \in [1, n]$ e $w \in [1, W]$:

```
DP[i][w] = max {
    DP[i-1][w],                       //(nao pegar item i)
    DP[i-1][w - peso[i]] + valor[i]   //(pegar item i)
}
```

Condição:

- A segunda opção só é válida se $peso[i] \leq w$.

Transições dos Subproblemas:
O problema DP[i][w] depende de:

- DP[i-1][w]: Melhor valor sem o item i (mesma capacidade, um item a menos)
- DP[i-1][w - peso[i]]: Melhor valor com capacidade reduzida para acomodar o item i

Essa estrutura caracteriza a subestrutura ótima: a solução ótima contém soluções ótimas dos subproblemas.

Solução Final: O valor ótimo está em DP[n][W].

3 Análise Empírica

3.1 Teste e resultados

- Capacidade da mochila: 50kg e 100 kg
- Valores de n testados: 10, 15, 25 (Força Bruta)
- Valores de n testados: 10, 15, 25, 50, 100, 200 (Guloso e DP)
- Pesos: Distribuição aleatória entre 5 e 14 kg
- Valores: Distribuição aleatória entre 27 e 72

Algoritmos	n = 10	n = 15	n = 25	n = 50	n = 100	n = 200
Força Bruta	0,00021s	0,00636s	2,15694s	———	———	———
Guloso	0,00001s	0,00001s	0,00001s	0,00005s	0,00007s	0,00013s
Programação Dinâmica	0,00001s	0,00001s	0,00001s	0,00007s	0,00013s	0,00019s

Gráficos:

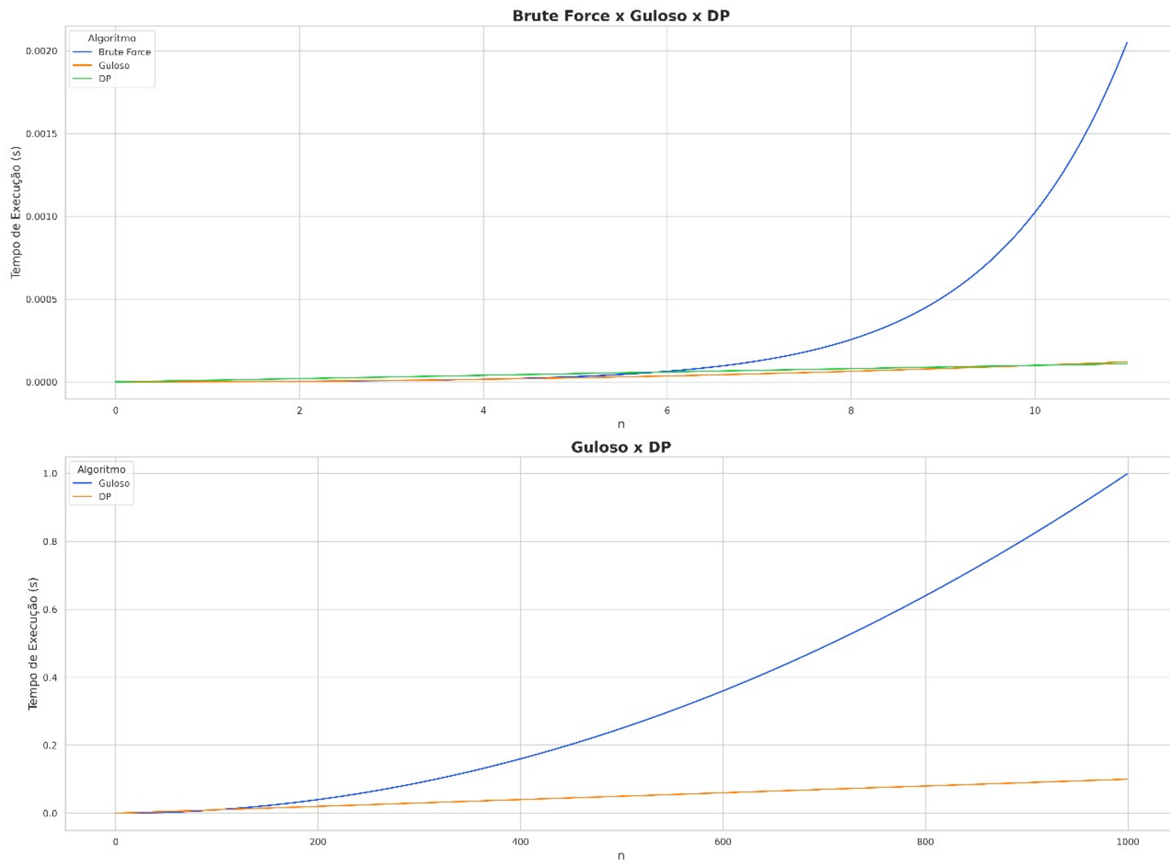

Figura 1: Gráficos comparativos entres os algoritmos.

Observações:

- Força Bruta cresce exponencialmente
- Ambos DP e Guloso crescem de forma aproximadamente linear (na escala log) no primeiro gráfico
- O crescimento de DP é mais lento que Guloso devido ao fator W
- Guloso tem crescimento quadrático puro: O(n^2)
- DP tem crescimento linear em n quando W é fixo

3.2 Discussão dos Resultados

Força Bruta:

- Teórico: $O(2^n)$
- Observado: Crescimento exponencial confirmado
- Tempo dobra aproximadamente a cada incremento de n
- Corrobora a análise teórica perfeitamente

Algoritmo Guloso:

- Teórico: $O(n^2)$

- Observado: Crescimento quadrático
- Para $n \times 10$, tempo $n \times 100$
- Resultados consistentes com a teoria

Programação Dinâmica:

- Teórico: $O(n \times W)$
- Observado: Crescimento linear (W fixo em 50 ou 100)
- Tempo proporcional a n quando W é constante
- Confirma dependência de ambos os parâmetros

4 Conclusões

Este trabalho implementou e analisou três abordagens clássicas para o Problema da Mochila 0/1:

Força Bruta: Garante solução ótima mas é inviável para instâncias grandes devido ao crescimento exponencial $O(2^n)$.

Algoritmo Guloso: Oferece solução rápida em O(n^2) mas não garante a solução ótima global. Adequado quando uma boa aproximação é suficiente.

Programação Dinâmica: Equilibra exatidão e eficiência com complexidade O(n×W). É a abordagem mais versátil para problemas de tamanho médio.