Universidade de São Paulo
Instituto de Ciências Matemáticas e Computação - ICMC

# Relatório do trabalho 1

Alunos: Augusto Ildefonso, Renan Trofino
Professor: Marcelo Garcia Manzato
Monitor: Lucas Padilha Modesto de Araujo

setembro
2024

Universidade de São Paulo
Instituto de Ciências Matemáticas e Computação - ICMC

# Relatório

Relatório do trabalho de Introdução à Ciência da Computação II.

Alunos: Augusto Ildefonso, Renan Trofino

Professor: Marcelo Garcia Manzato

Monitor: Lucas Padilha Modesto de Araujo

setembro
2024

# Sumário

# 1 Resumo

O presente relatório tem como objetivo explicar, implementar e comparar os algoritmos mais comuns para a solução do Problema da Mochila 0/1. Por meio de pesquisa e discussão de ideias, foram criadas soluções usando a linguagem C que resolvem o problema de forma eficiente e eficaz. As três soluções requisitadas envolvem algoritmo de força bruta (I), algoritmo guloso (II) e algoritmo com programação dinâmica (III).

# 2 Força Bruta

## 2.1 O algoritmo

Um algoritmo de força bruta é caracterizado por sua fácil implementação e elevada complexidade para grandes entradas. Ele se baseia na própria declaração do problema e nos conceitos envolvidos nele para resolvê-lo de forma simples.

Nesse problema, o algoritmo de força bruta gera todas as possíveis combinações de itens na mochila e compara o resultado dos valores obtidos, salvando o maior valor que se adequa ao peso máximo da mochila. Apesar de funcional, esse método é pouco eficiente para grandes entradas devido a alta possibilidade de combinações possíveis.

É importante ressaltar, até por questões de comparações com os outros algoritmos, que esse método gera todos os casos, o que o torna preciso. Além disso, o algoritmo implementado não faz uso de recursão, por isso não foi calculada a análise de recorrência.

## 2.2 Análise

A alta complexidade desse algoritmo decorre da geração de todas as possibilidades. Isso é expresso na implementação através dos loops for's. Fazendo a análise do algoritmo, percebe-se que a função $T(n)$ é $T(n) = 2^n$, pois ela possui dois loops for aninhados, o mais interno, no pior caso, será executado $N$ vezes, já o mais externo será executado $2^N$ (resultado da operação shift left aplicada para gerar o número de combinações).

Desse modo, tem-se a função $T(n) = N \times 2^N$, mas como $N$ é de ordem inferior à $2^N$, pode-se ignorá-lo, obtendo $T(n) = 2^n$. Assim, pode representar a complexidade do algortimo por $T(n) = O(2^n)$.

# 3 Guloso

## 3.1 O algoritmo

Um algoritmo guloso encontra ótimas soluções locais, na esperança de encontrar uma ótima solução global. O uso desse algoritmo torna o resultado aproximado, podendo ser ótimo.

Nesse processo, o algoritmo compara a importância de cada objeto, que é representada pela equação $imp = \frac{valor}{peso}$. O vetor de entrada será ordenado com base nesse cálculo, e irá mover os items de maior importância para o início da lista. Com a ordenação devidamente feita, basta adicionar na mochila o máximo de itens que for possível, sempre respeitando o peso máximo e ordem estabelecida para os itens.

Percebe-se que o algoritmo não avalia todos os casos, o que o torna impreciso em certos casos.

## 3.2 Análise

Para esse algoritmo, o processo mais custoso não envolve as ações da mochila, mas sim a ordenação do vetor que ocorre a priori dessa função. Dentro da organização da mochila existe apenas um laço, responsável por percorrer o vetor de itens.

Então, dependendo do tipo de algoritmo de ordenação, a complexidade pode variar. Para essa implementação (e por simplicidade) foi utilizado o bubble sort, que possui complexidade $O(n^2)$. Embora as melhorias estejam aplicadas, os casos beneficiados não estão nos extremos (melhor/pior caso), e sim nos valores intermediários.

# 4 Dinâmico

## 4.1 O algoritmo

Programação dinâmica é um método de desenvolvimento que envolve o uso de uma tabela para armazenar valores entre os diferentes paços da execução.

No problema da mochila, a tabela irá armazenar os valores de instâncias menores do problema, os sub problemas. A abordagem visa, justamente, entender e resolver os problemas menores para então chegar à solução desejada, o problema completo.

A tabela construída depende da capacidade C da mochila e do número N de items disponíveis. Usando dois laços aninhados, deve-se percorrer a tabela e realizar as comparações, retendo o valor resultante na posição correspondente de cada índice. A solução final será dada na última lacuna da tabela, quando os índices estiverem em sua última posição.

## 4.2 Análise

O algoritmo depende da tabela de N linhas e C colunas para operar, o que torna o uso de dois laços aninhados necessário.

A mais custosa na execução é, não coincidentemente, o bloco com os laços, que apresenta uma complexidade $O(NC)$.

Ressalta-se que não foi calculada a equação de recorrência, pois o algoritmo não faz o uso da recursão.

# 5 Análise dos Resultados

## 5.1 Análise de Complexidade

A análise de complexidade dos algoritmos será feita usando a notação $O$ (Big-Oh).

Para o algoritmo de força bruta, temos, como foi calculado anteriormente (já desprezando as tarefas de tempo constante), que a função $T(n)$ era $T(n) = 2^n$. Como a notação $O$ engloba casos em que os valores são menores ou iguais, pode-se usá-la sem maiores problemas. Então, supondo que $T(n) = O(2^n)$, verifica-se:

$$2^n \leq c\,2^n$$

Dividindo-se ambos os lados por $2^n$, isso pode ser feito pois não há valor de $n$ que zere essa potência:

$$1 \leq c$$

Assim, temos que para $\forall c \geq 1$, a igualdade a válida, o que permite afimar que $T(n) = O(2^n)$.

Com base nas análises de complexidade, percebe-se que o algoritmo de força bruta é o mais custoso, o que já era esperado por ser o que gera todas as possibilidades (através de um laço for que é executado $2^n$ o que é extremamente custoso).

O mais eficiente é o algoritmo de programação dinâmica, que constrói uma tabela de valores para depois encontrar o caso especificado no problema. A matriz possui dimensões variáveis, que dependem da quantidade de items $N$ e da capacidade de peso da mochila $C$. Por percorrer a tabela com dois laços, a complexidade depende do tamanho de cada laço (até $N$ e até $C$). Aqui, a complexidade será dada por $T(n, c) = nc$. Então têm-se $O(n\,c)$.

Por fim, o algoritmo guloso é o que apresenta resultados intermediários. Como não realiza todas as comparações entre items, tende a ser mais rápido. Porém, pelo mesmo motivo, seus resultados serão aproximados. O processo mais custoso é a ordenação dos items, que precede a criação da mochila (ação principal). Por questões de simplicidade, o algoritmo bubble sort foi utilizado para ordenar o vetor; Sua complexidade é $O(n^2)$ por conta de dois laços aninhados que percorrem o vetor em sua totalidade.

No ponto de vista das implementações, é clara a diferença das complexidades. Pelo laço mais complexo, o algoritmo de força bruta apresenta a maior complexidade, seguido do algoritmo guloso e por último, com menor complexidade, o algoritmo dinâmico. Então, a ordem decrescente de complexidade dos algoritmos é força bruta, guloso e programação dinâmica. Ou então: $O(2^n) > O(n^2) > O(n\,c)$.

## 5.2 Análise Empírica

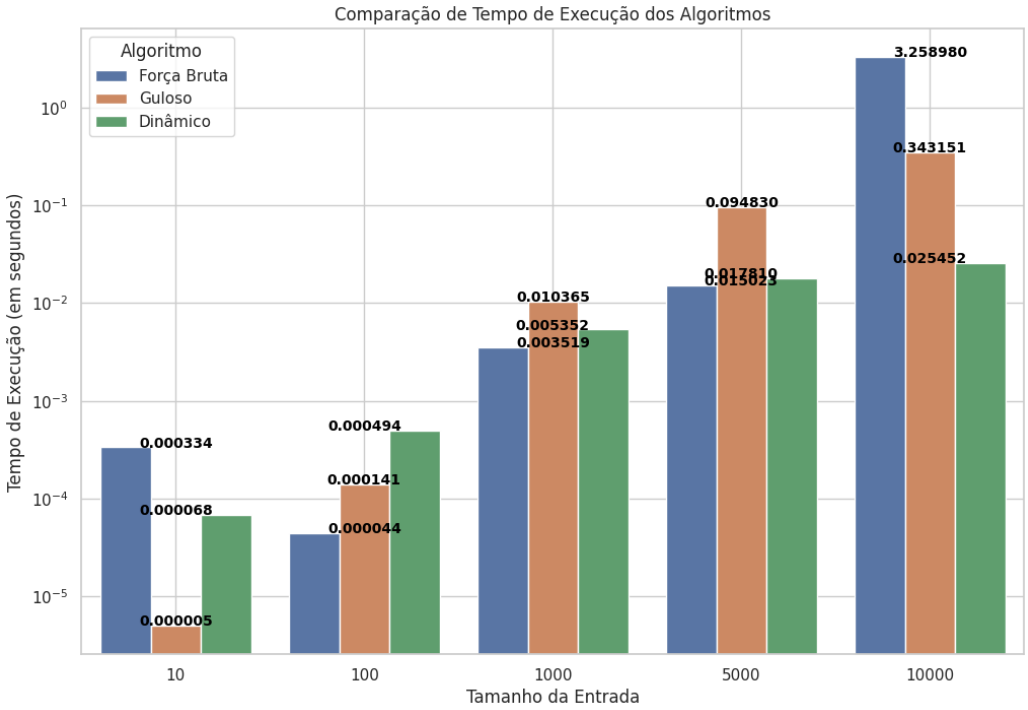


Figura 1: Comparação do tempos de execução dos algoritmos

| 10 | 100 | 1000 | 5000 | 10000 | Algoritmo |
|---|---|---|---|---|---|
| 0.000334 | 0.000044 | 0.003519 | 0.015023 | 3.258980 | Força Bruta |
| 0.000005 | 0.000141 | 0.010365 | 0.094830 | 0.343151 | Guloso |
| 0.000068 | 0.000494 | 0.005352 | 0.017810 | 0.025452 | Dinâmico |

Tabela 1: Tempos de execução (em segundos) nos diferentes casos teste

A partir dos dados acima, conclui-se que os algoritmos se comportam como o previsto, mantendo a ordem esperada de complexidade. Ou seja, o algoritmo menos eficiente é o da força bruta e o mais eficiente é o de programação dinâmica ($O(2^n) > O(n^2) > O(n\,c)$).

Ademais, a análise empírica permite notar que para entradas pequenas o algoritmo guloso foi o mais eficiente, o que faz sentido visto que a análise local dele serve perfeitamente para a análise geral, dado que a quantidade de dados pequena torna a região local próxima da geral.

É notória a mudança na execução dos algoritmos conforme o aumento do número de entradas. Como previsto, o algoritmo que utiliza programação dinâmica é, para casos grandes, o mais eficiente. Não existem saltos em seu tempo, o que mostra um aumento controlado e proporcional às entradas.

# Anexo

Todos os algoritmos, casos de teste e script para gerar gráficos estão disponíveis no repositório do projeto no Github (Disponível em `https://github.com/renan823/Trabalho-ICC2/`).

## 5.3 Código do Algoritmo de Força Bruta

```
/**
 * Código para resolver o problema da mochila com o método da força bruta
 *
 * @author Augusto Fernandes Ildefonso
 * @date 17/09/2024
 */
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

#define TAM_MAX 1000000 // Definindo o tamanho máximo da mochila

/**
 * Protótipos das funções
 */
int* forca_bruta(int peso[], int valor[], int N, int W, int mochila[]);
void zera_vetor(int** aux, int N);
void copia_vetor(int** mochila, int** aux, int N);

/**
 * Função que zera o vetor, é usada para gerar todas as possibilidades
 * de combinações dos itens
 *
 * @param aux Ponteiro para o vetor auxiliar
 * @param N Tamanho da mochila
 */
void zera_vetor(int** aux, int N){
    for(int i = 0; i < N; i++){
        (*aux)[i] = 0;
    }
}

/**
```

```
 * Função que copia o vetor aux para o vetor mochila
 *
 * @param mochila Ponteiro para o vetor mochila
 * @param aux Ponteiro para o vetor aux
 * @param N Tamanho da mochila
 */
void copia_vetor(int** mochila, int** aux, int N){
    for(int i = 0; i < N; i++){
        (*mochila)[i] = (*aux)[i];
    }
}

/**
 * Função que resolve o problema da mochila usando o método da força dupla
 *
 * @param peso Vetor que contém os pesos dos itens
 * @param valor Vetor que contém os valores dos itens
 * @param N Tamanho da mochila
 * @param W Peso máximo que a mochila suporta
 * @param mochila Vetor que representa a mochila
 *
 * @return Retorna a mochila que contém o maior valor, dentro do peso máximo
 */
int* forca_bruta(int peso[], int valor[], int N, int W, int mochila[]){
    int total_combinacoes = 1 << N;
    int peso_atual = 0, valor_atual = 0, maior_valor = 0;
    int peso_maior_valor;
    int* aux;

    aux = (int*) malloc(sizeof(int) * N);

    for(int i = 0; i < total_combinacoes; i++){
        zera_vetor(&aux, N);
        peso_atual = 0;
        valor_atual = 0;
        for(int j = 0; j < N; j++){
            if(i & (1 << j)){
                peso_atual += peso[j];
                valor_atual += valor[j];
                aux[j] = 1;
            }
```

```
        }

        if((peso_atual <= W) && (valor_atual > maior_valor)){
            maior_valor = valor_atual;
            peso_maior_valor = peso_atual;
            copia_vetor(&mochila, &aux, N);
        }
    }

    printf("\n\nValor total: %d\nPeso atual: %d\n", maior_valor,
    peso_maior_valor);
    return mochila;
}

int main(void){
    int N, W;
    int peso[TAM_MAX], valor[TAM_MAX], *mochila;

    scanf("%d %d", &N, &W);

    mochila = (int*) calloc(sizeof(int), N);

    for(int i = 0; i < N; i++){
        scanf("%d %d", &peso[i], &valor[i]);
    }

    //inicar contagem
    clock_t inicio = clock();

    mochila = forca_bruta(peso, valor, N, W, mochila);

    //fim da execução
    clock_t fim = clock();

    printf("Tempo decorrido: %lf segundos\n",
    (double)(fim - inicio) / CLOCKS_PER_SEC);

    printf("Mochila: (");
    for(int i = 0; i < N; i++){
        printf("%d", mochila[i]);
        if(i != N-1){
```

```
            printf(", ");
        }
    }
    printf(")\n");


    return(0);
}
```

## 5.4 Código do Algoritmo Guloso

```
/**
 * Código para resolver o problema da mochila usando programação dinâmica
 *
 * @author Renan Trofino Silva
 * @date 20/09/2024
*/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct item {
    int id;
    int valor;
    int peso;
} ITEM;

void bubble_sort(ITEM items[], int n) {
 ITEM aux;
 int trocas = 1;

 for (int i = 0; i < n && trocas > 0; i++) {
  trocas = 0;
  for (int j = 0; j < n-1-i; j++) {
            float valor1 = (float) items[j].valor/items[j].peso;
            float valor2 = (float) items[j+1].valor/items[j+1].peso;

   if (valor1 < valor2) {
    aux = items[j];
```

```
    items[j] = items[j+1];
    items[j+1] = aux;
    trocas++;
   }
  }
 }
}

/*
O algoritmo guloso tenta encontrar a melhor solução local.
Nesse caso, irá analisar a importância de cada objeto.
Ordenando o vetor de items pela importância.
Enquanto a mochila estiver vazia, coloque items.
E assim sucessivamente, até a mochila estar cheia.
*/
void mochila(ITEM items[], int n, int c, int* total, ITEM resultado[]) {
    int restante = c;

    //ordenar vetor
    bubble_sort(items, n);

    for (int i = 0; i < n; i++) {
        if (items[i].peso <= restante) {
            resultado[*total] = items[i];
            (*total)++;

            restante -= items[i].peso;
        }
    }
}

int main(void) {
    int N, C, pi, vi;

    scanf("%d %d", &N, &C);
    ITEM* items = (ITEM*) malloc(N * sizeof(ITEM)); //vetor de importância

    for (int i = 0; i < N; i++) {
        scanf("%d %d", &pi, &vi);
        items[i] = (ITEM){i, vi, pi};
    }
```

```
    ITEM* resultado = (ITEM*) malloc(N * sizeof(ITEM)); // Resultado
    int total = 0;

    //inicar contagem
    clock_t inicio = clock();

    mochila(items, N, C, &total, resultado);

    //fim da execução
    clock_t fim = clock();

    printf("Tempo decorrido: %lf segundos\n",
    (double)(fim - inicio) / CLOCKS_PER_SEC);

    printf("Mochila:\n");
    for (int i = 0; i < total; i++) {
        printf("Item %d: Valor = %d, Peso = %d\n", resultado[i].id,
        resultado[i].valor, resultado[i].peso);
    }

    free(items);
    free(resultado);

    return(0);
}
```

## 5.5 Código do Algoritmo de Programação Dinâmica

```
/**
 * Código para resolver o problema da mochila usando programação dinâmica
 *
 * @author Renan Trofino Silva
 * @date 17/09/2024
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/*
```

```
Cria uma matriz de inteiros usando os tamanhos especificados.
A matriz possui ponteiros para ponteiros para int.

As dimensões são passadas por parâmetro (colunas e linhas, respectivamente)

A matriz alocada é retornada.
Em caso de erro, o porgrama é encerrado.
*/
int** criar_matriz(int n, int c) {
    int** matriz = (int**) malloc(n * sizeof(int*));
    if (matriz == NULL) {
        exit(1);
    }

    for (int i = 0; i < n; i++) {
        matriz[i] = (int*) calloc(c, sizeof(int)); //já inicia tudo zerado!
        if (matriz[i] == NULL) {
            exit(1);
        }
    }

    return(matriz);
}

/*
Aloca um vetor dinâmicamente.
O número de items e o tamanho (sizeof) do item são parâmetros.
O vetor é retornado.
*/
void* criar_vetor(int items, int tamanho) {
    void* vetor = (void*) malloc(items * tamanho);
    if (vetor == NULL) {
        exit(1);
    }

    return(vetor);
}

/*
Libera a memória de uma matriz alocada.
Antes de liberar os ponteiros para ponteiros,
```

```
libera-se os ponteiros que representam as linhas.
*/
void apagar_matriz(int*** matriz, int n) {
    for (int i = 0; i < n; i++) {
        free((*matriz)[i]);
        (*matriz)[i] = NULL;
    }

    free(*matriz);
    *matriz = NULL;
}

/*
Exibe uma matriz, linha por linha
*/
void matriz_print(int** matriz, int n, int c) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < c; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }
}

int max(int v1, int v2) {
    if (v1 > v2) {
        return(v1);
    }

    return(v2);
}

/*
Os números são naturais (isso é muito importante)

Uma matriz (tabela) com N linhas e C colunas guarda
os valores de cada sub problema (uma parte menor)

N -> número de items
C -> peso máximo
```

```
Se N = 0 ou C = 0 => Nada será feito

O vetor P armazena os pesos
O vetor V armazena os valores

Usando índices i e j, pode-se definir o item matriz[i][j]
como o sub problema evolvendo P, V i e j.

O item matriz[i][j] representa os priemiros "i" items
cujo peso somado não ultrapassa "j".

Para matriz[0][j] (nenhum item) => matriz[0][j] = 0

Existem dois casos para analisar matriz[i][j]:
    Se o peso P[i] > j:
        matriz[i][j] = matriz[i-1][j]
    Senão:
        O valor matriz[i][j] será o valor máximo
        envolvendo matriz[i-1][j] e V[i] + matriz[i-1][j-P[i]]

O primeiro caso ocorre quando o peso do novo item é maior que o limite.
Encontrar a carga máxima, ou seja, maximizar a os valores, é encontrar o item
da matriz que seja matriz[N][C]

Esse algoritmo é relativamente parecido com o algoritmo de otimização de troco!

A complexidade do algoritmo não depende só de N, mas também de C (capaciade).
Isso ocorre pois a tabela é a principal ação, e a tabela tem formato N * C.

O acesso i-1 é usada pois i = 1 e j = 1 no laço (mas não nos vetores P e V)
*/
void mochila(int p[], int v[], int n, int c, int** resultado, int* total) {
    int** matriz = criar_matriz(n+1, c+1);

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= c; j++) {
            if (p[i-1] > j) {
                matriz[i][j] = matriz[i-1][j]; //não cabe
            } else{
                matriz[i][j] = max(matriz[i-1][j],
                v[i-1] + matriz[i-1][j-p[i-1]]);
```

```
            }
        }
    }

    int j = c;
    for (int i = n; i > 0; i--) {
        if (matriz[i][j] == matriz[i-1][j]) {
            //caso oposto ao de cima (não cabe)
            (*resultado)[i-1] = 0;
        } else {
            (*resultado)[i-1] = 1;
            j -= p[i-1];
            //foi usado -> coloque no resultado
        }
    }

    *total = matriz[n][c];

    apagar_matriz(&matriz, n+1);
}

int main(void) {
    int N, C;

    scanf("%d %d", &N, &C);

    int* P = (int*) malloc(N * sizeof(int)); //vetor pesos
    int* V = (int*) malloc(N * sizeof(int)); //vetor valores

    for (int i = 0; i < N; i++) {
        scanf("%d %d", &P[i], &V[i]); //inserir peso e valor de cada item
    }

    int vTotal = 0;
    int pTotal = 0;
    int *res = (int*) criar_vetor(N, sizeof(int));

    //inicar contagem
    clock_t inicio = clock();

    mochila(P, V, N, C, &res, &vTotal);
```

```
    //fim da execução
    clock_t fim = clock();

    printf("Tempo decorrido: %lf segundos\n",
    (double)(fim - inicio) / CLOCKS_PER_SEC);

    printf("Mochila: (");
    for(int i = 0; i < N; i++){
        printf("%d", res[i]);
        if(i != N-1){
            printf(", ");
        }

        if (res[i] == 1) {
            pTotal += P[i];
        }
    }
    printf(")\n");

    printf("Valor total: %d\n", vTotal);
    printf("Peso total: %d\n", pTotal);

    free(P);
    free(V);
    free(res);

    return(0);
}
```

# Referências


[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*. MIT press, 2022.

[2] Feofillof, P. *Algoritmos: Em Liguagem C*. Elsevier Brasil, 2013.

[3] Feofiloff, P. Análise de algoritmos. *Internet: http://www. ime. usp. br/pf/analise_de_algoritmos 2009* (1999).