Projeto 1 - Problema da Mochila 0/1

Lucas Dúckur Nunes Andreolli - N° USP: 15471518
Rafael Perez Carmanhani - N° USP: 15485420

1 Introdução

O problema da Mochila 0/1 é um problema clássico de computação que possui o seguinte enunciado:

> Dado um conjunto de n itens, cada um com um peso w_i e um valor v_i, e uma mochila que pode suportar um peso máximo W, determine a combinação de itens que maximiza o valor total sem exceder a capacidade da mochila. Cada item pode ser incluído ou excluído da mochila na sua totalidade (não é permitido fracionar itens).

Para a resolução do problema, três algortimos foram confeccionados de acordo com três paradigmas diferentes: o paradigma de **Força Bruta**; o **Algortimo Guloso**; e a **Programação dinâmica**. Os três algoritmos foram avaliados teoricamente, por meio da análise de complexidade, e também empiramente, por intermédio de medições de tempo de execução de acordo com diferentes entradas para n e W.

2 Força bruta

2.1 Definição

O paradigma de Força Bruta é um método de resolução de problemas no qual todos os candidatos a solução são avaliados e aquele que atender ao enunciado do problema é escolhido. No caso do problema da Mochila 0/1, isso significa avaliar o valor total de todas as possíveis combinações de itens tais que o peso máximo não seja ultrapassado e, em seguida, escolher o maior dentre eles.

2.2 Implementação

O algoritmo implementado que segue o paradigma de Força Bruta é um algoritmo recursivo: a cada chamada, ele avalia se o n-ésimo item deve ou não ser incluído na mochila, de modo que todos os subcojuntos de itens são avaliados. Uma vez que todas as possíveis configurações da mochila são verificadas, essa resolução encontra sempre a melhor solução. Implementação em C do algoritmo:

```
/* Funcao que resolve o problema da Mochila 0/1 segundo o paradigma da Forca Bruta.
A funcao recebe como parametros um vetor de inteiros "v", que armazena os valores de
cada item; um vetor de inteiros "p", que armazena os pesos de cada item; um inteiro
```

```
"limite" que representa o peso maximo da mochila; e um inteiro "n", que representa o
numero de items.*/

int forca_bruta(int v[], int limite, int p[], int n) {
1     //Caso base: n = 0 ou limite = 0.
2     //Nesse caso, eh impossivel incluir qualquer item
3     if(n == 0 || limite == 0)
4        return(0);
5
6     //Se o n-esimo item tiver peso maior
7     //do que o limite, nao o inclua
8     if(p[n - 1] > limite)
9        return(forca_bruta(v, limite, p, n - 1));
10
11    //Retorna o maior valor dentre dois casos:
12    //n-esimo termo eh incluso ou nao
13    int res = max(v[n - 1] + forca_bruta(v, limite - p[n - 1], p, n - 1),
14                    forca_bruta(v, limite, p, n - 1));
15    return(res);
16 }
```

Na linha 13, é usada a função *max*, que calcula o valor máximo entre dois inteiros. Implementação em C da função *max*:

```
/*Funcao que calcula o maior valor entre dois inteiros
Ela recebe como parametros dois inteiros a e b. */
int max (int a, int b) {
1  if(a > b)
2     return(a);
3  else
4     return(b);
5  }
```

2.3 Análise de Complexidade

Para determinar a complexidade do algoritmo, considera-se o caso de entrada que demanda o maior tempo de execução. No contexto atual, isso ocorre quando cada chamada da função realiza outras duas chamadas recursivas (linhas 13 e 14 da função *forca_bruta*) em vez de apenas 1 (linha 8), isto é, quando todos os itens forem selecionados, uma vez que, nesse caso, as recursões das linhas 13 e 14 ocorrem todas as vezes e o caso base ocorre apenas quando $n = 0$. Assim, para se determinar a equação de recorrência do algoritmo, deve-se considerar a seguinte recursão: para se solucionar cada problema de tamanho n, é necessário um tempo constante (para as comparações) e é preciso resolver dois subproblemas de tamanho $n - 1$, até o caso base em que $n = 0$.

Desse modo, a equação de recorrência do tempo $T(n)$ é:

$$T(n) = \begin{cases} O(1), & \text{se } n = 0 \\ 2T(n-1) + O(1), & \text{caso contrário} \end{cases}$$

Como cada chamada da função realiza outras duas chamadas, faz-se a hipótese de que a solução para a recorrência é $O(2^n)$. Prova-se essa hipótese pelo método de substituição, usando a hipótese

de indução de que $T(n) < c2^n - d$, em que $d = O(1)$:

- **Caso base**: quando $n = 0$, $T(n) = d < c2^0 - d$ para $c > 2d$.
- **Passo indutivo**: suponha que a hipótese é verdadeira para $n_0 = 1 \leq k < n$. Então:

$$\begin{aligned} T(n) &= 2T(n-1) + d \\ &< 2c2^{n-1} - 2d + d \\ &= c2^n - d \end{aligned}$$

Portanto, escolhendo $c > 2d$ e $n_0 = 1$, tem-se que $T(n) < c2^n - d \ \ \forall n > n_0$. Logo, está provado que $T(n) = O(2^n)$.

2.4 Análise empírica

Para se verificar o comportamento do algoritmo empiricamente, foram realizados testes de medição de tempo de execução para valores de n tais que $10 \leq n \leq 25$. Ao longo dos testes, os pesos e valores dos itens foram gerados aleatoriamente (uso da função *rand()*), bem como o limite de peso W, de forma a garantir a veracidade dos testes. A partir de $n \geq 25$, torna-se imprático fazer testes sucessivos, uma vez que cada teste leva vários segundos. Os resultados obtidos encontram-se no seguinte gráfico:

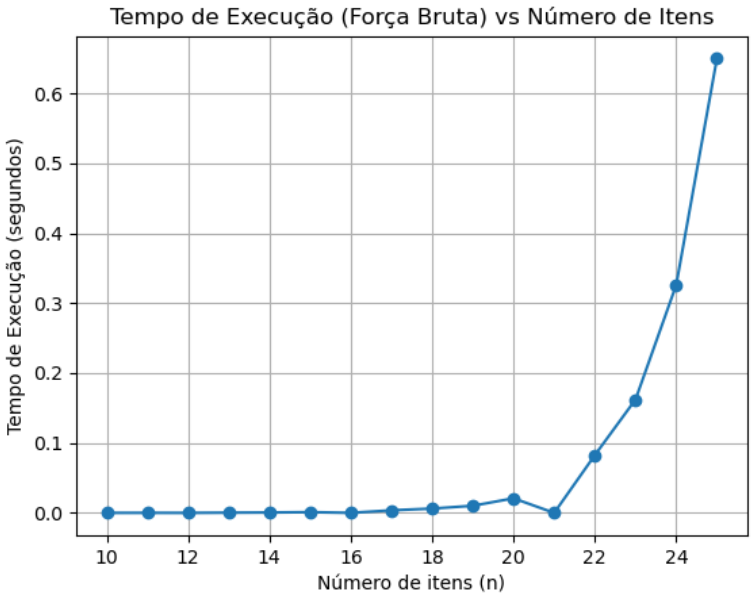

Figura 1: análise empírica do algoritmo Força Bruta

O gráfico obtido reflete o comportamento assintótico teórico do algoritmo: uma vez que ele é $O(2^n)$, espera-se que seu gráfico seja uma curva exponencial, isto é, espera-se que, a partir de um dado valor de n, os valores do tempo de execução tendam ao infinito rapidamente. Isso se verifica nos

testes feitos, haja vista que, a partir de $n = 21$ (Figura 1), a curva passa a crescer de forma bastante acelerada. Evidencia-se assim a similaridade entre o formato do gráfico apresentado e o gráfico da função exponencial $f(n) = 2^n$:

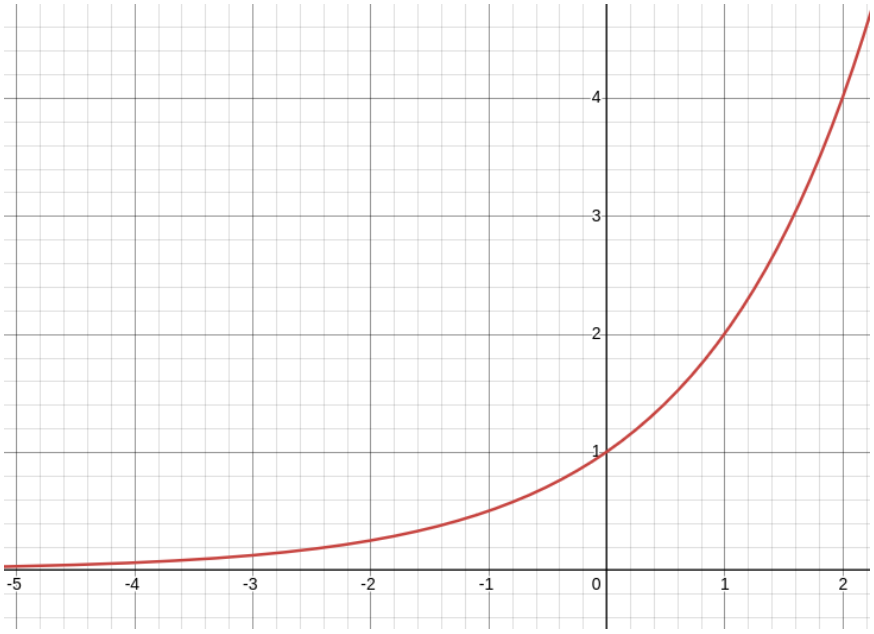

Figura 2: gráfico da função $f(n) = 2^n$ no intervalo $[-5, 2]$

3 Algoritmo Guloso

3.1 Definição

O Algortimo Guloso é um paradigma de resolução de problemas que toma decisões localmente ótimas com o objetivo de se atingir a solução global. No contexto da Mochila 0/1, isso significa escolher os itens com maiores razões valor/peso e incluí-los na mochila até que esta esteja cheia ou os itens acabem.

Como é impossível fracionar itens no problema da Mochila 0/1, essa resolução não obtém sempre a solução ótima - em alguns casos, o valor na mochila é maximixado sem que todos os itens escolhidos sejam os de melhor relação valor/peso. Um exemplo simples em que isso ocorre é na seguinte configuração de três itens:

Valores: $(60, 100, 120)$ - *Pesos:* $(10, 20, 30)$ - *Peso máximo:* 50.

As razões valor/peso, portanto, são: $(6, 5, 4)$.

O algoritmo guloso escolheria os itens 1 e 2 e não escolheria o item 3, uma vez que isto excederia o peso máximo. Assim, o valor obtido seria 160. Contudo, o valor máximo é obtido ao se escolher os itens 2 e 3 e não o item 1, resultando em um valor máximo verdadeiro de 220.

3.2 Implementação

O algoritmo implementado é iterativo: ele se utiliza de uma *struct* "elemento" para armazenar os itens e suas respectivas razões valor/peso; em seguida, ele ordena esses itens em ordem decrescente de razão e os coloca na mochila até que a capacidade máxima seja atingida. Implementação em C:

```
/* Funcao que resolve o problema da Mochila 0/1 segundo o paradigma do Algoritmo Guloso.
A funcao recebe como parametros um vetor de inteiros "v", que armazena os valores de
cada item; um vetor de inteiros "p", que armazena os pesos de cada item; um inteiro
"limite" que representa o peso maximo da mochila; e um inteiro "n", que representa o
numero de items.*/
int algoritmo_guloso(int v[], int limite, int p[], int n) {
1  //Criacao e inicializacao de um vetor que
2  //armazena cada item
3  elemento vet[n];
4  for(int i = 0; i < n; i++) {
5     vet[i].razao = (double) v[i]/p[i];
6     vet[i].valor = v[i];
7     vet[i].peso = p[i];
8  }
9
10 //Ordenacao dos itens de acordo
11 //com a razao valor/peso
12 quick_sort(vet, 0, n - 1);
13
14 //escolha dos itens com melhor relacao
15 // valor/peso ate que a mochila esteja
16 //cheia ou que acabem os itens
17 int peso = 0, valor = 0;
18 for(int k = 0; k < n; k++) {
19    if(peso + vet[k].peso > limite)
20       continue;
21
22    valor += vet[k].valor;
23    peso += vet[k].peso;
24    if(peso == limite)
25       break;
26 }
27
28 return(valor);
29 }
```

Implementação da *struct* "elemento" (linha 3):

```
    /* Struct que armazena os itens. */
typedef struct {
1  int valor; //valor do item
2  int peso; //peso do item
3  double razao; //razao valor/peso do item
} elemento;
```

O algoritmo utilizado para ordenar os itens (linha 12 da função *algoritmo_guloso*) é o *QuickSort*. Implementação do *QuickSort* em C:

```
/*Funcao que implementa o algoritmo de ordenacao
QuickSort para ordenar um vetor de itens de acordo
com a razao valor/peso */
void quick_sort(elemento v[], int inf, int sup) {
1  int meio = (inf + sup)/2;
2  double pivo = v[meio].razao;
3
4  int i = inf;
5  int j = sup;
6
7  do {
8     while(v[i].razao > pivo) i++;
9     while(v[j].razao < pivo) j--;
10
11    if(i <= j) {
12       elemento aux = v[i];
13       v[i] = v[j];
14       v[j] = aux;
15       i++;
16       j--;
17    }
18 }while(i < j);
19
20 if(j > inf) quick_sort(v, inf, j);
21 if(i < sup) quick_sort(v, i, sup);
22 }
```

3.3 Análise de Complexidade

A complexidade do algoritmo implementado depende de duas etapas: a primeira é a ordenação do vetor de itens (linha 12 da função *algoritmo_guloso*); a segunda é a seleção dos itens (linhas 17-26). Note que a segunda etapa, no pior caso, leva um tempo $T(n) = n$; logo, ela possui um comportamento assintótico igual a $O(n)$. Isso implica que o comportamento assintótico do algoritmo é igual ao da segunda etapa.

O algoritmo de ordenação utilizado é o *QuickSort*. Sua complexidade, no pior caso, é de $O(n^2)$. No entanto, a escolha do pivô como sendo o elemento do meio do vetor (linha 2 da função *quick_sort*) reduz as chances do pior caso ocorrer. Assim, deve-se considerar a complexidade do caso médio do algoritmo *QuickSort* para a análise, que é $O(n \log n)$. Portanto, a complexidade $T(n)$ do algoritmo guloso é:

$$T(n) = O(n \log n) + O(n) + O(1) = \underline{O(n \log n)} \quad (\textit{caso médio})$$

3.4 Análise Empírica

Foram realizados testes de medição de tempo de execução idênticos aos realizados para a análise do algoritmo de Força Bruta. No caso, os valores de n são tais que $100000 \leq n \leq 500000$. Acima

disso, problemas relativos ao uso excessivo de memória começam a surgir. Os resultados obtidos encontram-se no seguinte gráfico:

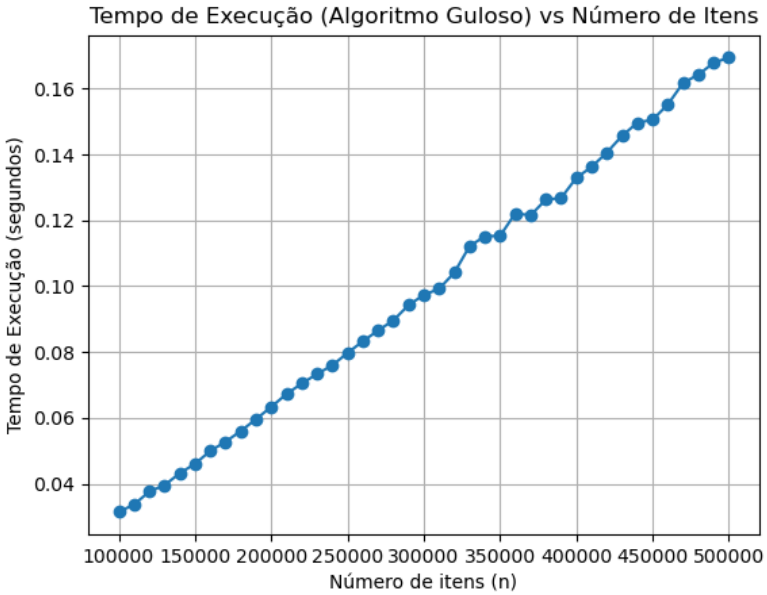

Figura 3: análise empírica do Algoritmo Guloso

Como a complexidade do programa é dependente da complexidade do algoritmo de ordenação utilizado, espera-se que a curva acima se assemelhe à curva que representa o comportamento assintótico teórico do caso médio do algoritmo *QuickSort*, que é $f(n) = n \log n$. Contudo, devido à baixa taxa de crescimento de $\log n$, o gráfico *tempo de execução* x n só irá se diferenciar visualmente de uma curva linear para valores bem maiores de n e por tempos de execução maiores. Verifica-se a similaridade visual do gráfico de $f(n) = n \log n$ com uma reta na curva a seguir:

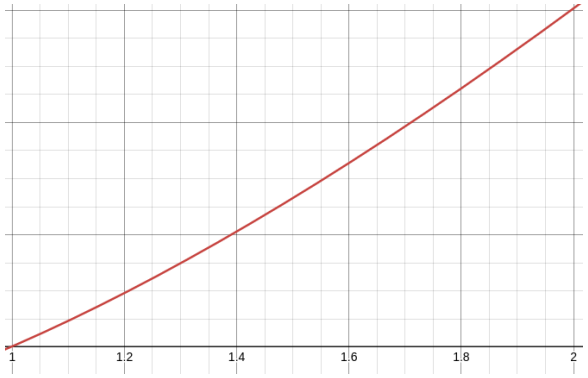

Figura 4: Gráfico da função $f(n) = n \log n$ no intervalo $[1, 2]$

4 Programação Dinâmica

4.1 Definição

Programação dinâmica é uma técnica de resolução de problemas computacionais em que problemas complexos são divididos em subproblemas menores, que são resolvidos e cujos resultados são armazenados de modo a evitar recomputações. No contexto do problema da Mochila 0/1, isso significa resolver os subproblemas definidos pelos primeiros i itens e por uma capacidade $w < W$.

Uma vez que o algoritmo encontra soluções ótimas para cada subproblema definido por i e w até que $i = n$ e $w = W$, ele sempre encontra a solução ótima para o problema.

4.2 Implementação

O algoritmo implementado é iterativo: ele cria e preenche uma tabela bidimensional que armazena o resultado de todos os subproblemas. Assim, ele é composto por dois laços de repetição encadeados, sendo que o mais externo itera o valor de i enquanto que o mais interno itera o valor de w até que $i = n$ e $w = W$, de modo que os resultados de todas as combinações i/w são calculados, armazenados na tabela e usados para calcular os resultados das combinações seguintes. Implementação em C:

```
/* Funcao que resolve o problema da Mochila 0/1 segundo o paradigma da Programacao
Dinamica.A funcao recebe como parametros um vetor de inteiros "v", que armazena os
valores de cada item; um vetor de inteiros "p", que armazena os pesos de cada item;
um inteiro "limite" que representa o peso maximo da mochila; e um inteiro "n", que
representa o numero de items.*/
int programacao_dinamica(int v[], int limite, int p[], int n) {
1  int tabela[n + 1][limite + 1];
2
3  //Preenchimento da tabela que
4  //armazena os resultados de cada
5  // subproblema i/w
6  for(int i = 0; i <= n; i++) {
7     for(int w = 0; w <= limite; w++) {
8        //Se nao for possivel incluir nenhum
9        //item ou o peso maximo for zero,
10       //o valor maximo eh zero
11       if(i == 0 || w == 0)
12          tabela[i][w] = 0;
13       else{
14
15       if(p[i - 1] > w)
16          //Se o peso do ultimo item que pode ser incluso for
17          //maior que o peso maximo, o valor maximo eh igual ao
18          //valor maximo dos i - 1 itens com o mesmo limite w
19          tabela[i][w] = tabela[i - 1][w];
20       else
21
```

```
22          //Se o item puder ser incluso, o valor maximo eh o maximo entre o valor
23          //de quando ele eh incluso e de quando ele nao o eh
24          tabela[i][w] = max(tabela[i - 1][w], tabela[i - 1][w - p[i - 1]] + v[i - 1]);
25       }
26    }
27 }
28
29 return(tabela[n][limite]);
30 }
```

Na linha 24, é usada a função *max*, definida na implementação do algoritmo de Força Bruta.

4.3 Análise de Complexidade

O algoritmo é composto por dois laços de repetição encadeados: o mais externo (linha 6) itera $n+1$ vezes e,a cada iteração, o mais interno (linha 7) itera $W+1$ vezes, o que mostra que o tempo de execução do algoritmo depende do valor de n e de W. Dessa forma, considerando os tempos constantes de comparação, atribuição e da função *max*, o tempo de execução $T(n)$ é igual a:

$$\begin{aligned} T(n) &= (n+1)\cdot(W+1) + O(1) \\ &= nW + n + W + 1 + O(1) \end{aligned}$$

Como $nW \geq n$ e $nW \geq W$ para todo n, W (sendo n e W inteiros positivos):

$$\underline{T(n,W) = O(n \cdot W)}$$

4.4 Análise Empírica

Foram realizados testes de tempo de execução para valores de n tais que $0 \leq n \leq 1000$. Contudo, a complexidade do algoritmo depende do peso máximo W. Logo, esses testes foram realizados considerando $W = 500$, $W = 1000$ e $W = 1500$. Valores maiores para n e W dificultam os testes na medida em que causam problemas de memória. Os resultados são sumarizados no seguinte gráfico:

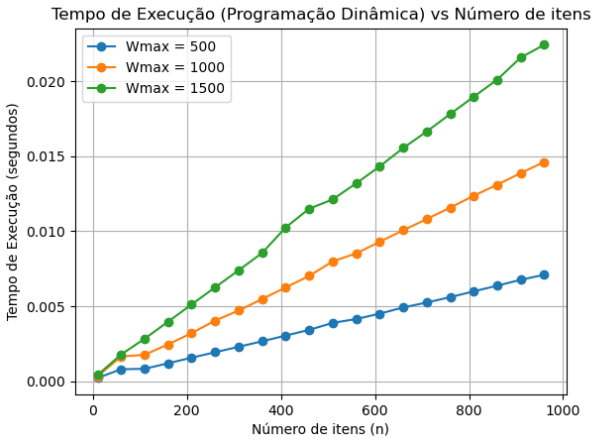

Figura 5: análise empírica do algoritmo que segue Programação Dinâmica

O gráfico apresentado evidencia o comportamento assintótico teórico do algoritmo: se W for mantido constante, o gráfico é uma reta que representa a função $f(n) = Wn$. Mais ainda, como o algoritmo é $O(W \cdot n)$, os tempos de execução do programa são diretamente proporcionais a W, como se evidencia pela diferença entre as três curvas do gráfico nas quais $W = 500$, $W = 1000$ e $W = 1500$.

5 Comparações entre os algoritmos

Teoricamente, o algoritmo de Força Bruta leva muito mais tempo para ser executado, uma vez que sua complexidade é $O(2^n)$, ou seja, exponencial. Mais ainda, para valores baixos de n, a complexidade média do algoritmo *QuickSort* se aproxima muito de uma complexidade linear devido ao baixo crescimento de $\log n$; desse modo, espera-se que o comportamento assintótico do Algoritmo Guloso seja aproximadamente linear para n pequeno. Assim, em comparação com o algoritmo de Programação Dinâmica, cuja complexidade é a de um termo linear multiplicado por W - $O(n \cdot W)$ -, ele apresenta tempos de execução menores. Portanto, a classificação crescente (teórica) em termos de tempo de execução para n pequeno é: Algoritmo Guloso < Programação Dinâmica < Força Bruta.

Essa ordem foi verificada empiricamente por meio de testes de tempo de execução para $0 \leq n \leq 30$ (n maiores implicam tempos de execução tão grandes para o algoritmo de Força Bruta que dificultam a medição), considerando que $W = 1000$. Foi utilizada uma escala logarítmica para representar os resultados, haja vista que as diferenças enormes entre os resultados do algoritmo de Força Bruta e os dos outros dois dificulta a visualização. Os resultados são representados a seguir:

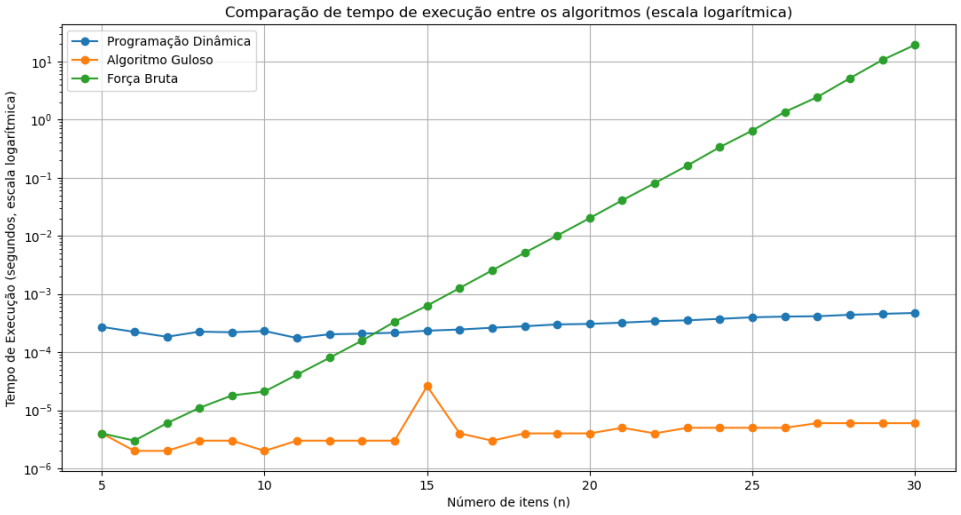

Figura 6: comparação empírica entre os algoritmos implementados

É evidente que, a partir de $n = 15$, o algoritmo de Força Bruta torna-se mais lento do que os outros,

como o esperado. Ademais, a razão entre os tempos de execução do algoritmo de Programação Dinâmica e o Algoritmo Guloso permance constante (devido aos detalhes teóricos explicados) e aproximadamente igual a 1000, que corresponde ao valor W que multiplica o fator linear na complexidade do programa que segue o paradigma da Programação Dinâmica.

6 Conclusão

A partir das análises empíricas e teóricas, conclui-se que o algoritmo de Força Bruta é aplicável apenas para valores pequenos de n. Ademais, para valores menores desse parâmetro e a depender do valor do peso máximo W, o Algoritmo Guloso é geralmente mais rápido do que o de Programação Dinâmica, apesar de nem sempre alcançar a solução ótima. O algoritmo de Programação Dinâmica, por sua vez, atinge sempre a melhor solução e é muito mais eficiente do que o algoritmo de Força Bruta, sendo, portanto, recomendável para valores de n para os quais há memória suficiente para se implementar esse algoritmo.