Relatório de CTC-12 / 2023

Laboratório 04 - Paradigmas de programação

Aluno

Marcel Versiani e Silva

Turma COMP.25

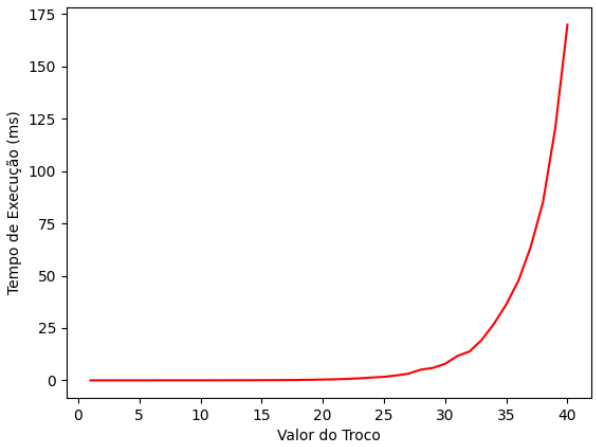
Professor

Luiz Gustavo Bizarro Mirisola

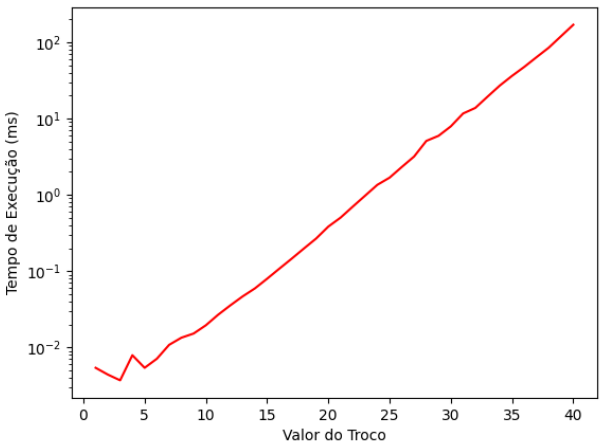
Instituto Tecnológico de Aeronáutica - ITA

1 Problema das Moedas de Troco

1.1 Gráficos dos resultados dos algoritmos

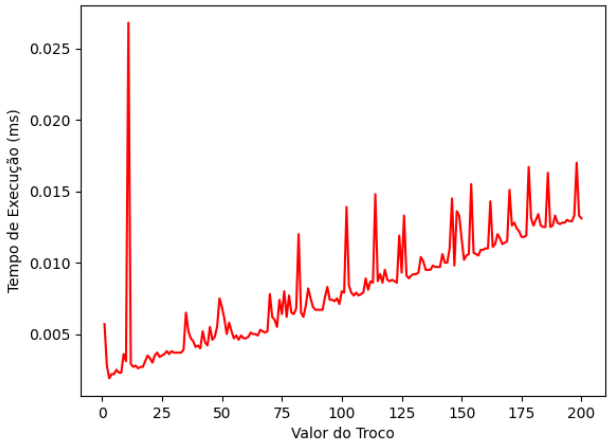


(a) Algoritmo de Divisão e Conquista (DC)

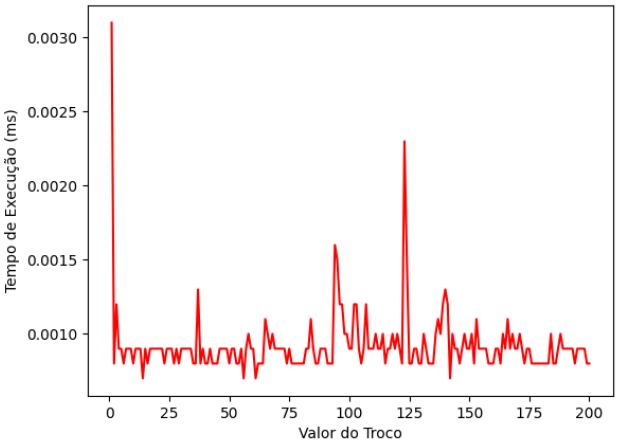


(b) Algoritmo DC (Escala logarítmica)

Figura 1: Tempos de Execução de Algoritmos



(a) Algoritmo de Programação Dinâmica (PD)



(b) Algoritmo Guloso

Figura 2: Tempos de Execução de Algoritmos

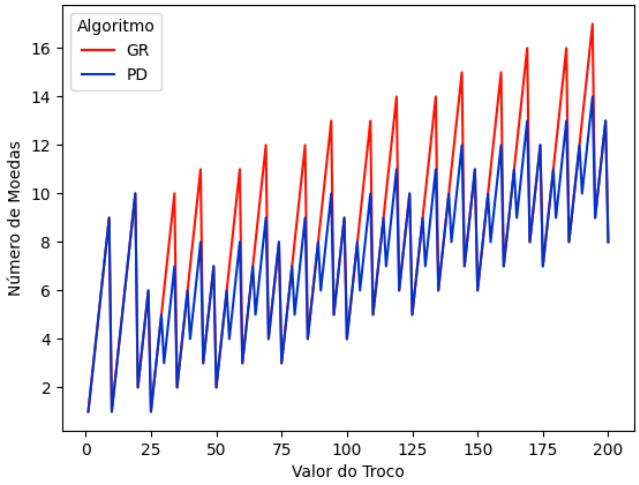


Figura 3: Número de Moedas Retornadas pelo Algoritmos GR e PD

1.2 Velocidade e otimização dos algoritmos

A partir dos gráficos das Figuras 1 e 2, percebe-se que a complexidade de tempo dos algoritmos de Divisão e Conquista, Programação Dinâmica e Guloso foram, respectivamente, $\Theta(2^n)$, $\Theta(n)$ e $\Theta(1)$, relevando os *outliers* de que aparecem ao longo de alguns valores de trocos.

Esse comportamento era esperado, pois o algoritmo DC calcula todos os subcasos possíveis para valores crescentes de 1 até *value*, superpondo vários casos de troco iguais e com isso realiza vários cálculos desnecessários, obtendo uma complexidade de tempo exponencial (Figura 1b). Ademais, as chamadas recursivas utilizam a pilha de execução além das chamadas adicionais de função, e aloca um vetor auxiliar a cada chamada, o que piora o comportamento do algoritmo.

O algoritmo de programação dinâmica também calcula os subscasos de 1 até *value*, mas os subcasos não são superponíveis, então a complexidade de tempo é linear (Figura 2a). Isso se deve ao fato de que ele armazena os valores ótimos de cada subcaso em vetores auxiliares para utilizar depois, evitando cálculos desnecessários que aparecem no DC.

O algoritmo guloso depende apenas do tamanho do vetor de moedas disponíveis *denom* que, dado constante para vários valores de *value*, exibe então uma complexidade de tempo constante (Figura 2b), mesmo que em alguns casos ele seja mais lento que o PD por causa das constantes. Com isso, observa-se que o algoritmo guloso é ótimo em termos de complexidade de tempo, contudo, não é ótimo em termos de números de moedas utilizadas e em alguns casos pode até não resolver um problema que possui solução ($denom = [10, 25]$ e $value = 40$).

A partir do gráfico da Figura 3, pode-se observar que o algoritmo guloso, a partir de uma certa quantidade de valor de troco, utiliza mais moedas do que o necessário, sendo que o de divisão e conquista utiliza sempre o número mínimo de moedas (solução ótima). Com isso, os algoritmos DC e PD são ótimos no sentindo de número de moedas, pois calculam subcasos com quantidade ótima e usam essas informações para calcular o número mínimo de moedas do *value* em questão.

1.3 Discussão do algoritmo de Divisão e Conquista

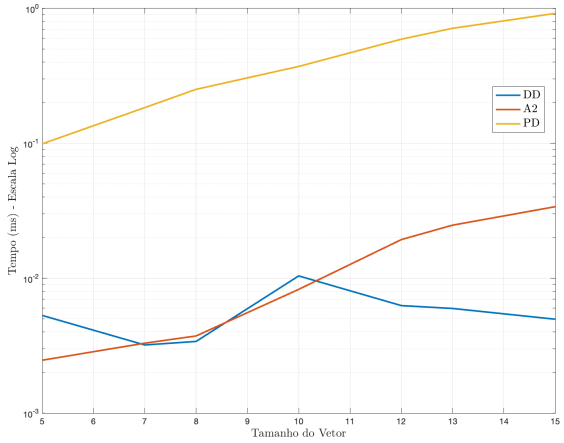
De fato, o algoritmo DC foi o mais lento dentre os três, contudo essa limitação é devida a especificidades da estrutura do problema em questão, não do paradigma de programação.

Pode-se pegar um exemplo como o algoritmo *MergeSort*, que utiliza a divisão e conquista para ordenar vetores com base em comparações, e com isso consegue ser ótimo em termos de complexidade de tempo para essa estrutura de problema. Neste caso, o subcasos não se superpõe e o *merge* utiliza totalmente as informações dos subcasos para ordenar os vetores que estão cada vez mais perto da raiz da árvore de empilhamento, ou seja, não realiza cálculos desnecessários.

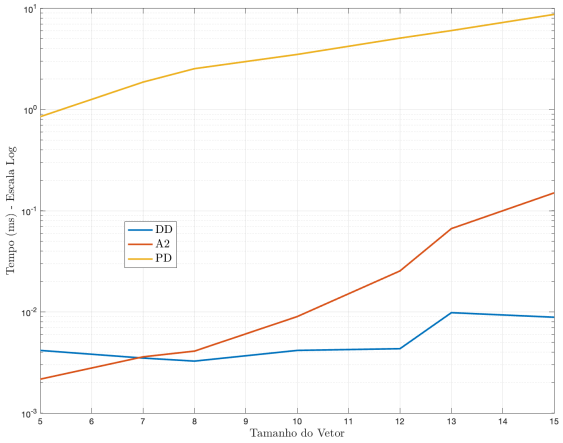
No caso das Moedas de Troco, a implementação desse paradigma leva ao cálculo de subcasos iguais várias vezes, ou seja, existem casos superponíveis e com isso a resposta do problema não utiliza várias informações já calculadas. Com isso, tal especificidade do problema gera um código que, utilizando esse paradigma, não leva a uma complexidade de tempo ótima, mas consegue achar sempre o número mínimo de moedas necessárias.

2 Problema da Mochila

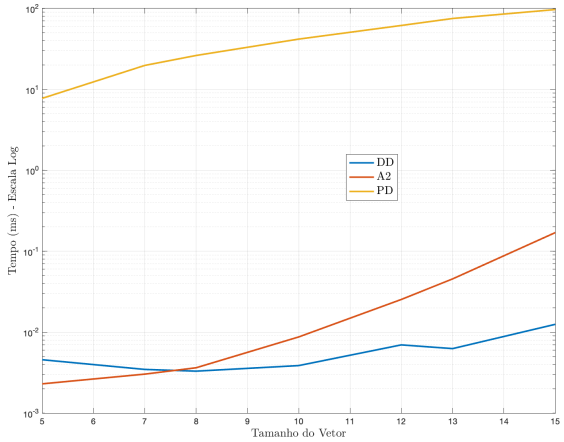
2.1 Gráfico dos resultados dos algoritmos



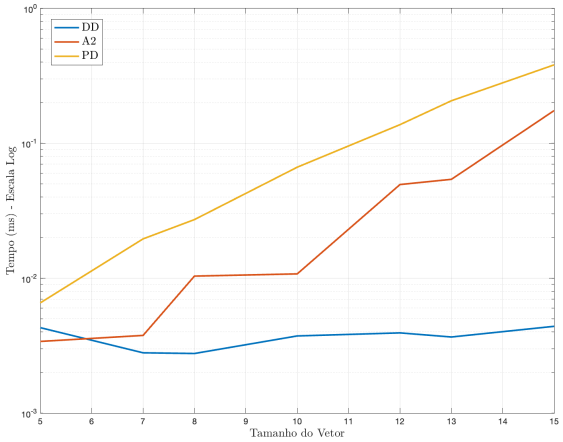
(a) Teste P3



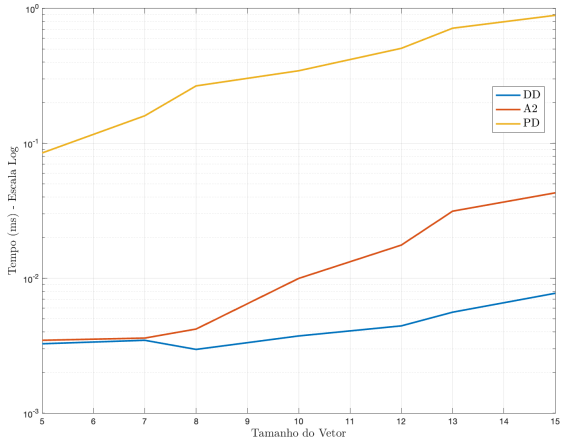
(b) Teste P4



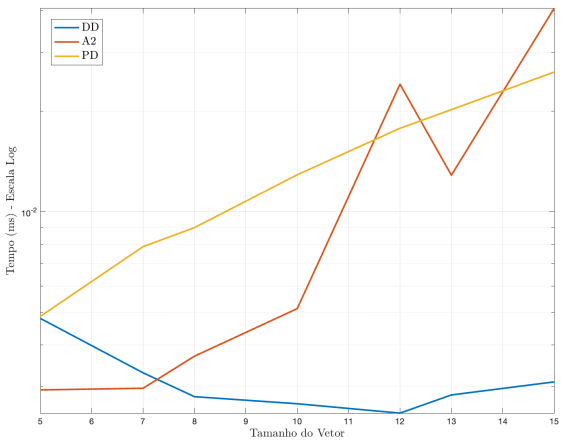
(c) Teste P5



(d) Teste AVIS



(e) Teste EVOD



(f) Teste RAND

Figura 4: Tempos de Execução de Algoritmos (Escala Logarítmica)

2.2 Algoritmo de Programação Dinâmica e melhorias

A ideia do algoritmo é inicializar a tabela $B_{n,w}$, sendo n o tamanho o *input* e w o *value*, com zeros correspondendo a não selecionar nenhum item e, após isso, completar cada linha em ordem crescente da esquerda para direita. Cada entrada da tabela corresponde a considerar os itens de 1 até k na linha k para $k > 0$ e a capacidade máxima w na linha w. A tabela é preenchida de modo a obter subsoluções dos problemas para que essa informação seja usada nos problemas maiores e verificar se a adição de um novo item faz sentido para obter um solução nova. A complexidade de tempo desse algoritmo é dado por $\Theta(nw)$, que é o número de elementos da matriz B utilizada.

No algoritmo da Programação Dinâmica não foram feitas melhorias pois foi utilizado o exemplo da sala de aula com algumas modificações nos índices da matriz para melhor se adequar ao problema.

2.3 Algoritmo *Meet-in-the-Middle* (A2) e melhorias

A ideia do algoritmo escolhido é, em vez de contar todas as possibilidades de soma dos subconjuntos do *input* ($\Theta(2^n)$) e verificar a soma desejada, divide-se o vetor em duas metades e calculam-se as possíveis somas dos subconjuntos desses subvetores ($\Theta(2^{\frac{n}{2}})$ e, para um desses vetores de somas, procura-se no outro ($\Theta(n2^{\frac{n}{2}})$) a diferença do *value* e, com isso, encontra-se a união de dois subconjuntos que resultam na soma desejada.

A melhoria nesse algoritmo foi usar *bit shift* e *bit mask* para que os índices dos vetores de soma, em binário, guardassem a informação de quais elementos do vetor foram utilizados na soma. Perceba que os vetores das somas têm tamanho 2^n, sendo n o número de elementos do vetor utilizado para calcular a soma dos conjuntos, ou seja, os índices em binário possuem n bits. Então se usarmos um *bit shift* de j zeros para a esquerda do número 1 em binário, sendo j o índice do elemento no vetor, pode-se usar a *bit mask* com o índice da soma para saber se aquele elemento está contido nessa soma.

Com isso, basta saber quais valores somam o *value* em cada vetor de somas que, dados os índices, sabemos quais elementos do *input* foram utilizados. Essa melhoria economiza tempo porque as operações *bit* a *bit* são rápidas e estamos usando uma informação que está presente inerentemente na estrutura de dados utilizada, que é um vetor, diminuindo as constantes multiplicativas na complexidade de tempo.

2.4 Velocidade dos algoritmos

Percebe-se que para praticamente todos os casos em todas as instâncias de teste, o algoritmo *Meet-in-the-Middle* obteve um melhor desempenho em relação ao PD. Mesmo que a complexidade de tempo do PD seja melhor, as constantes multiplicativas e os *values* utilizado são tais que o MM obteve um melhor desempenho, até melhor que o gabarito para vetores pequenos.

Para o caso aleatório (RAND), o MM tende a ser pior que o PD para vetores maiores porque os elementos podem ser grandes o suficiente tal que as várias operações de soma dos subconjuntos (exponencial) gastam mais tempo de processamento, aumentando as constantes multiplicativas e fazendo com que a implementação usada não seja tão interessante nesse casos. Para os casos que não têm solução, o MM tende a ser o pior pois são realizados todos os cálculos de soma possíveis e a busca em cada um deles, enquanto que o PD tende a ser mais rápido pela utilização da tabela que dá informações de casos anteriores para resolver o caso atual.