



Relatório de CES-12

Lab 04 – Moedas de Troco & SSP

Aluno: Fernando de Moraes Rodrigues

Turma Comp-22

Professor Luiz Gustavo Bizarro Mirisola

Instituto Tecnológico de Aeronáutica – ITA

Julho de 2020

MOEDAS DE TROCO

1) Resultados, para cada valor de troco, de tempo de execução e número de moedas de troco

- Divisão-e-Conquista (DC)

Esse algoritmo consiste em dividir o problema inicial em subproblemas análogos e disjuntos, utilizando resoluções recursivas e, por fim, fazendo uma combinação entre as subsoluções encontradas.

Para o problema das moedas, o tempo de execução (em ms) em função do valor do troco pode ser visto na figura 1 a seguir. Na figura 2, temos a quantidade de moedas utilizadas em função do valor do troco para esse problema.

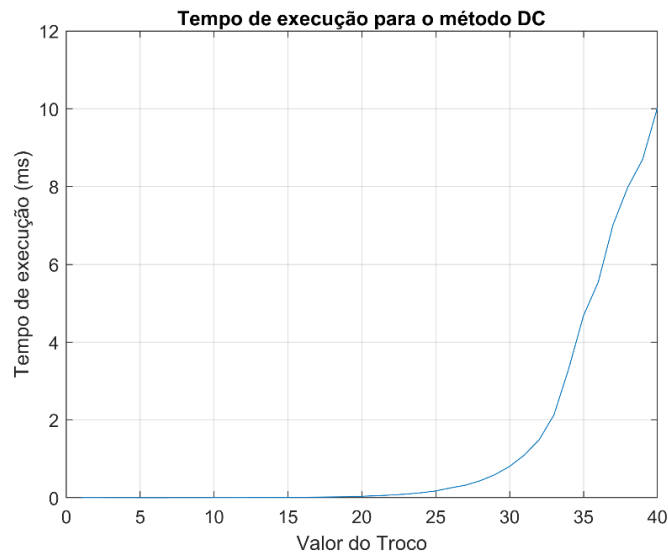


Figura 1. Tempo de execução em função do valor do troco para o método divisão-e-conquista

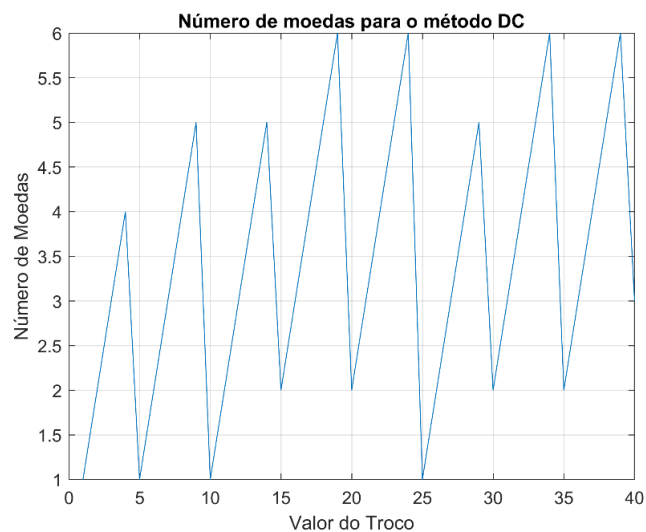


Figura 2. Quantidade de moedas utilizadas em função do valor do troco para o método divisão-e-conquista

- **Método *Greedy* (GR)**

Esse algoritmo baseia-se na extensão de soluções parciais (solução incremental), escolhendo a que propicia o maior ganho imediato (otimização de um critério local). Quando funciona, origina algoritmos simples e eficientes.

Para o problema das moedas, o tempo de execução (em ms) em função do valor do troco pode ser visto na figura 3 a seguir. Na figura 4, temos a quantidade de moedas utilizadas em função do valor do troco para esse problema.

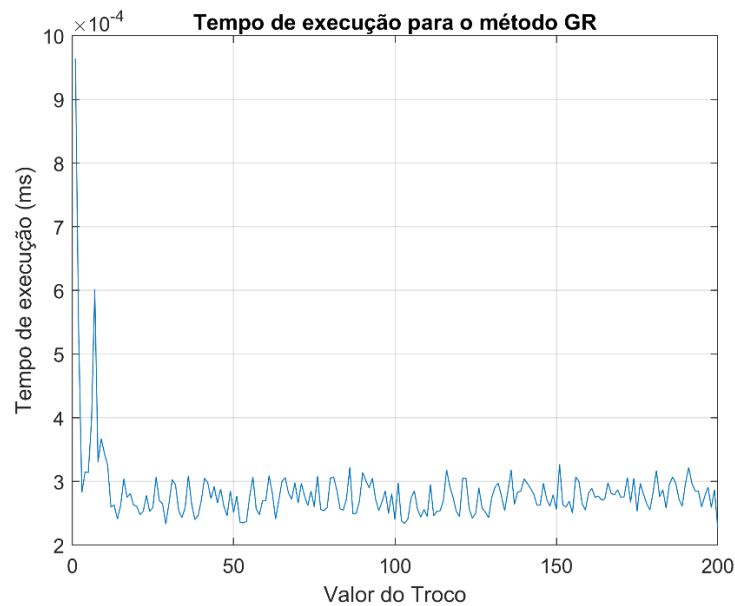


Figura 3. Tempo de execução em função do valor do troco para o método greedy

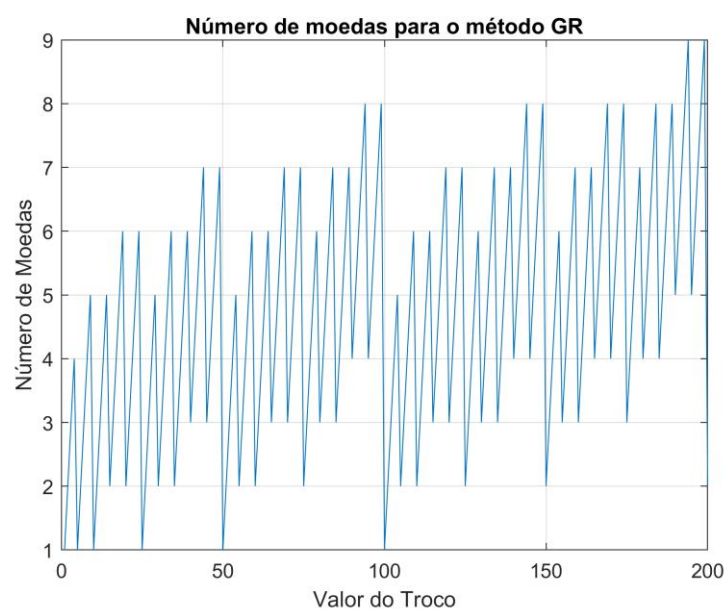


Figura 4. Quantidade de moedas utilizadas em função do valor do troco para o método greedy

- Programação Dinâmica (PD)

Esse algoritmo consiste em dividir o problema inicial em subproblemas análogos, mas sobrepostos, diferente do divisão-e-conquista. Em seguida, os subproblemas são resolvidos em ordem crescente de tamanho, armazenando os resultados em uma única tabela. Com isso, há um ganho de tempo por resolver-se uma única vez cada subproblema, mas há um aumento no espaço de armazenamento.

Para o problema das moedas, o tempo de execução (em ms) em função do valor do troco pode ser visto na figura 5 a seguir. Na figura 6, temos a quantidade de moedas utilizadas em função do valor do troco para esse problema.

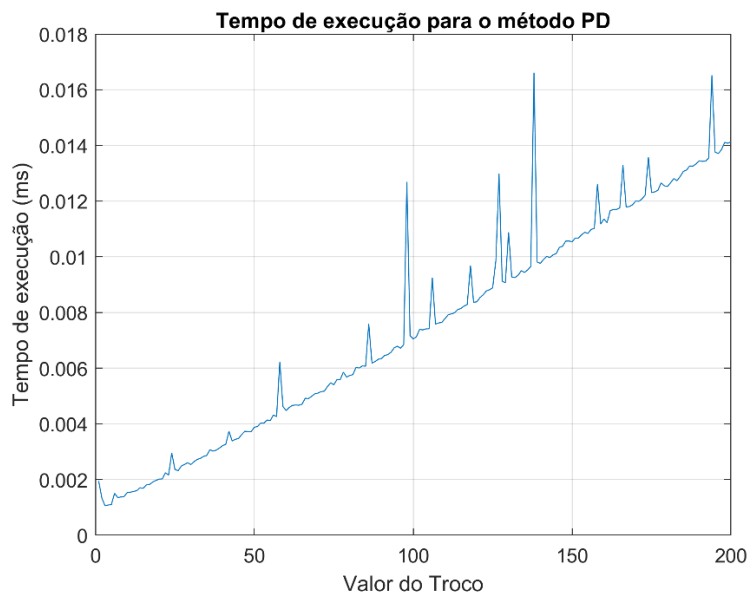


Figura 5. Tempo de execução em função do valor do troco para a programação dinâmica

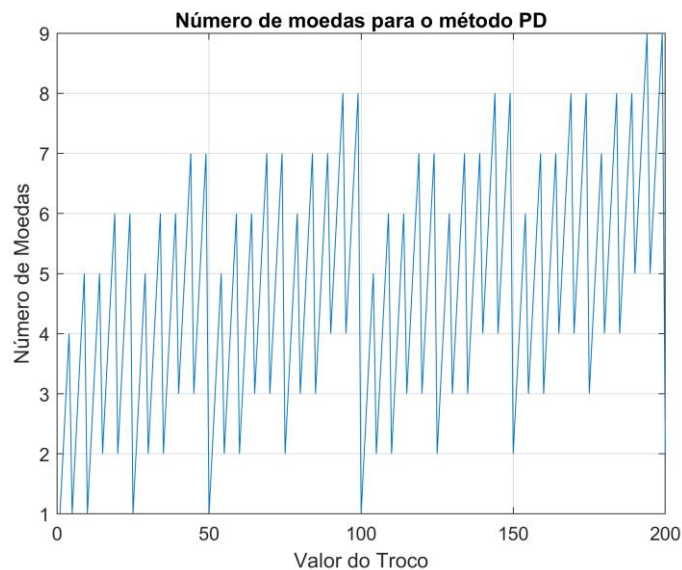


Figura 6. Quantidade de moedas utilizadas em função do valor do troco para a programação dinâmica

2) Explicação de porque os algoritmos são mais rápidos ou mais lentos, e porque são ótimos ou não, considerando as diferenças entre os paradigmas e como estas diferenças se aplicam na estrutura do problema específico

- Tempo

Quanto às ordens dos tempos de execução, vistos nas figuras 1, 3 e 5 respectivamente e descontando eventuais ruídos, temos que divisão-e-conquista é de ordem exponencial, método *greedy* é de ordem constante e programação dinâmica é de ordem linear.

Quanto ao tempo de execução em si, a divisão-e-conquista atinge tempos de execução muito elevados para uma quantidade pequena de valor de troco devido ao seu comportamento exponencial, proporcionado pelo crescente número de chamadas recursivas, como pode ser visto na figura 7 a seguir. Assim, dos três métodos, o DC apresenta o pior desempenho em tempo de execução.

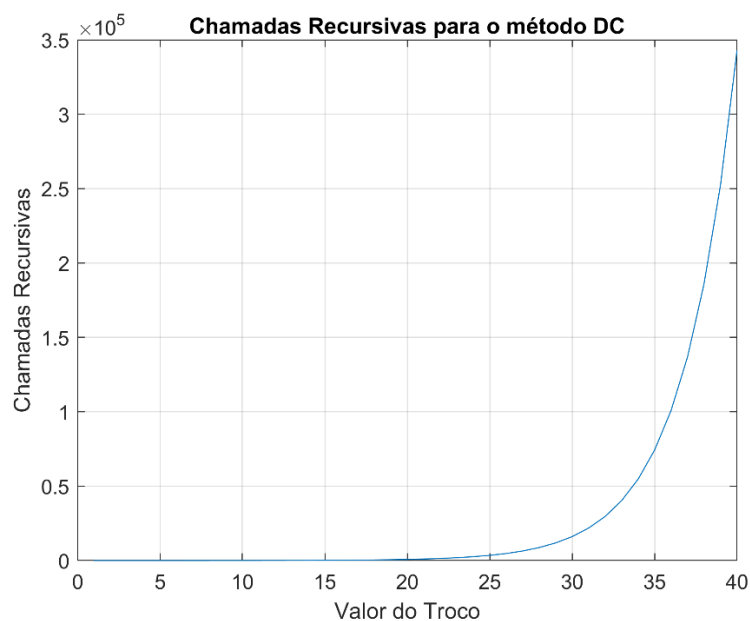


Figura 7. Quantidade de chamadas recursivas para o método divisão-e-conquista

Note que para um valor do troco de 40 a quantidade de chamadas recursivas já sobe consideravelmente, o que eleva bastante o tempo de execução desse método, como visto na figura 1 anteriormente. Por outro lado, os outros dois métodos calculam a quantidade de moedas em um tempo bem menor para valores de troco de até 200, como visto nas figuras 3 e 5 anteriormente.

Quanto à comparação dos outros dois métodos, temos que o método *greedy* (GR) tem complexidade de tempo $\Theta(k)$ (constante), sendo k o número de denominações, pois no pior caso serão analisadas todas as denominações. Já a programação dinâmica (PD) tem complexidade de tempo $\Theta(n.k)$ (linear), onde n é o valor do troco, pois no pior dos casos serão analisados todos os valores de troco e também todas as denominações.

Portanto, os tempos de execução crescem na ordem $GR < PD < DC$.

- Otimalidade

Quanto à otimalidade, tanto a divisão-e-conquista (DC) como a programação dinâmica (PD) apresentam soluções ótimas, pois sempre verificam nos casos anteriores e comparam para saber se há uma solução melhor (menor quantidade de moedas), tornando seus tempos de execução maiores que do *greedy* (GR).

Para o *greedy* nem sempre se tem soluções ótimas. Sabendo que a programação dinâmica dá soluções ótimas e há divergências no gráfico da figura 8 a seguir, podemos afirmar que o *greedy* nem sempre dará soluções também ótimas.

Essa não otimalidade do método *greedy* ocorre pois esse método termina a busca quando encontra uma solução que satisfaça o troco desejado, mesmo que a quantidade de moedas não seja a mínima possível. Isso explica também o fato de esse método ser mais rápido, pois não explora todas as possibilidades até encontrar a que seria ótima.

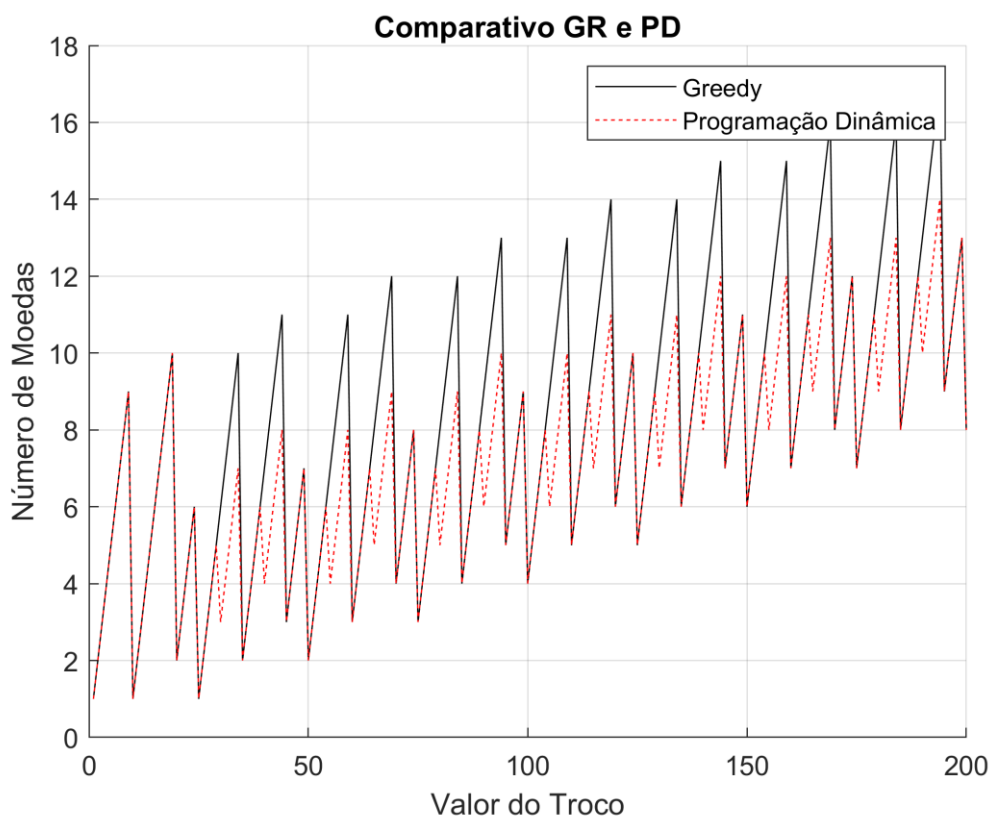


Figura 8. Comprovação da divergência entre os métodos GR e PD a partir de certo valor de troco

3) O algoritmo de Divisão-e-Conquista é claramente mais lento do que os outros. Mas isso é devido a uma limitação inerente ao paradigma em geral, ou se deve a alguma especificidade na estrutura do problema em questão?

Forneça um exemplo de problema que pode ser resolvido rapidamente por Divisão-e-Conquista e detalhe porque a estrutura deste problema permite essa solução rápida enquanto o nosso problema não permite.

O algoritmo de divisão-e-conquista utilizado no *MergeSort* e no *QuickSort* para o problema da ordenação apresenta complexidade de tempo ótima. Se fosse utilizada programação dinâmica não seria tão eficaz, pois seriam armazenadas subsoluções que não seriam consultadas novamente, uma vez que na ordenação não se resolve mais de uma vez um mesmo problema.

No nosso problema, a resolução de problemas repetidos é algo recorrente, o que sugere o uso da programação dinâmica de modo a obter uma solução ótima.

Portanto, a lentidão da divisão-e-conquista para o problema das moedas de troco deve-se a uma especificidade do problema em si, e não a uma limitação do paradigma, que pode obter desempenho ótimo em algoritmos de ordenação, por exemplo.

SSP – SUBSET SUM PROBLEM

1) Descrição da Implementação

- Programação Dinâmica (PD)

Seja o vetor de inteiros *input* de tamanho *n*, ou seja, os índices variam de 0 a *n* – 1, contendo os pesos dos itens a serem adicionados na mochila, *value* a capacidade da mochila e *B[k][w]* uma matriz que guarda a solução ótima dos *k* primeiros itens e uma mochila de capacidade *w*. Quando *w* = *value*, temos a solução do problema.

Para calcular *B[k][w]*, devemos verificar a relação entre *input[k]* e *w*. Caso *input[k]* > *w*, o item não pode ser adicionado e *B[k][w]* = *B[k-1][w]*. Caso contrário, *B[k][w]* será o máximo valor entre *B[k-1][w]* e *B[k-1][w – input[k]] + input[k]*. O processo se repete utilizando dois comandos *for* até *w* = *value* e *k* = *n* – 1. Se *B[k][w]* = *value* haverá solução.

Esse algoritmo foi implementado com base nos slides do professor Alonso.

- Backtracking (BB)

Esse algoritmo foi implementado fazendo uma busca em profundidade em uma árvore binária. Cada “nó” conterá dois valores: um referente ao valor da soma acumulada até o momento e outro com a capacidade restante da mochila. Caso o valor da capacidade seja negativo ou tenhamos chegado a uma folha mas a mochila ainda não está em sua capacidade máxima, devemos buscar em outro ramo de modo recursivo.

Na figura 9 a seguir temos um exemplo de como funciona o algoritmo.

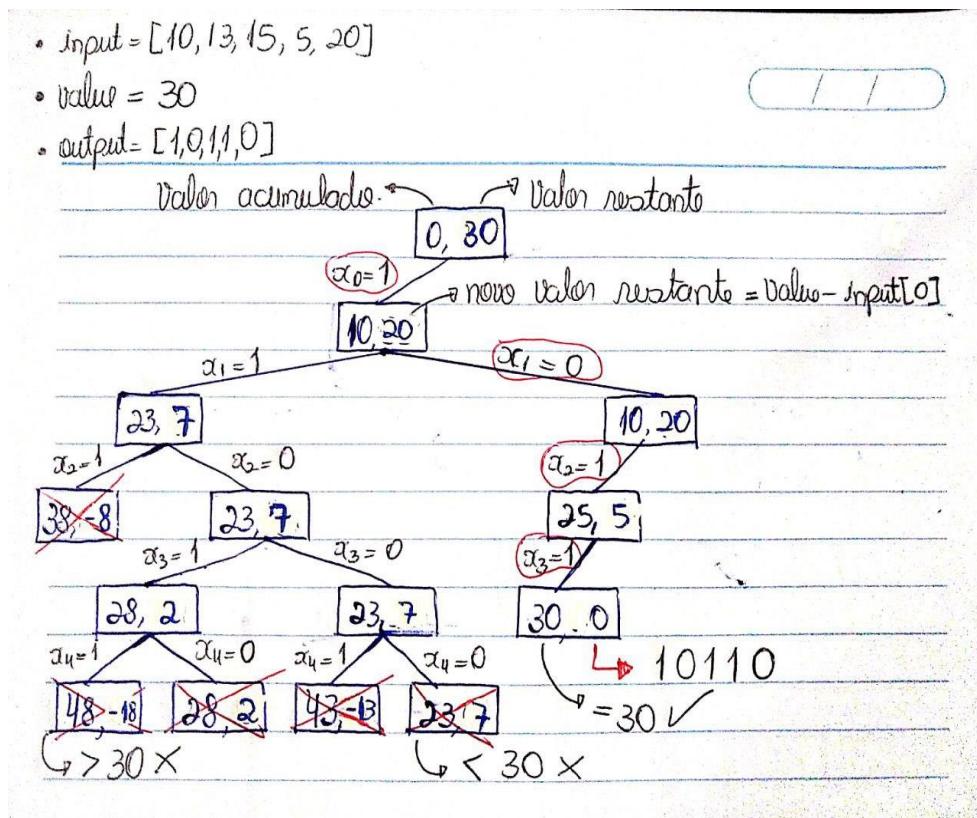


Figura 9. Exemplo do backtracking

Esse algoritmo foi implementado com base nos slides fornecidos junto com o roteiro e no vídeo do link <https://www.youtube.com/watch?v=kyLxTdsT8ws&t=200s>

2) Comparação dos Resultados

As figuras 10, 11, 12, 13, 14 e 15 a seguir mostram os resultados do tempo de execução em função do tamanho da entrada (input) para todos e cada um dos geradores de instâncias (AVIS, Even/Odd, P3, P4, P5 e Rand respectivamente), sendo o comparativo feito entre BB (backtracking), DD (gabarito) e PD (programação dinâmica).

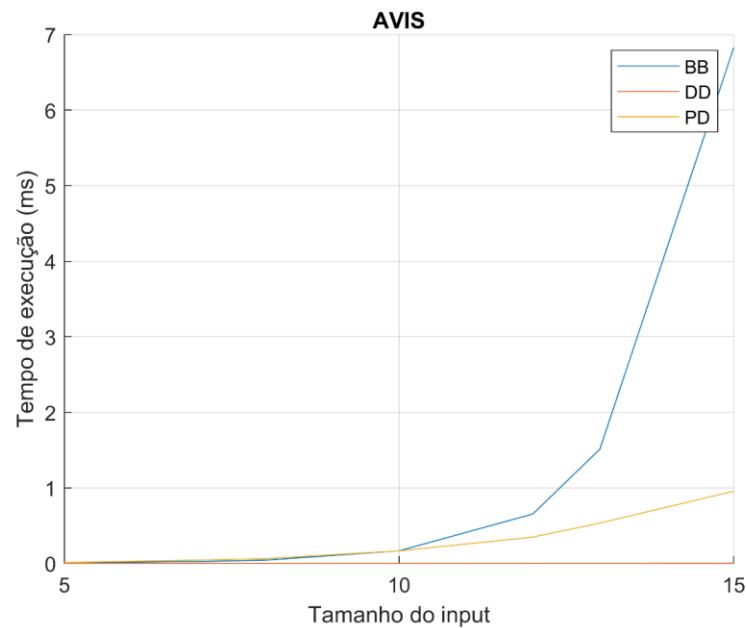


Figura 10. Tempo de execução para a instância AVIS

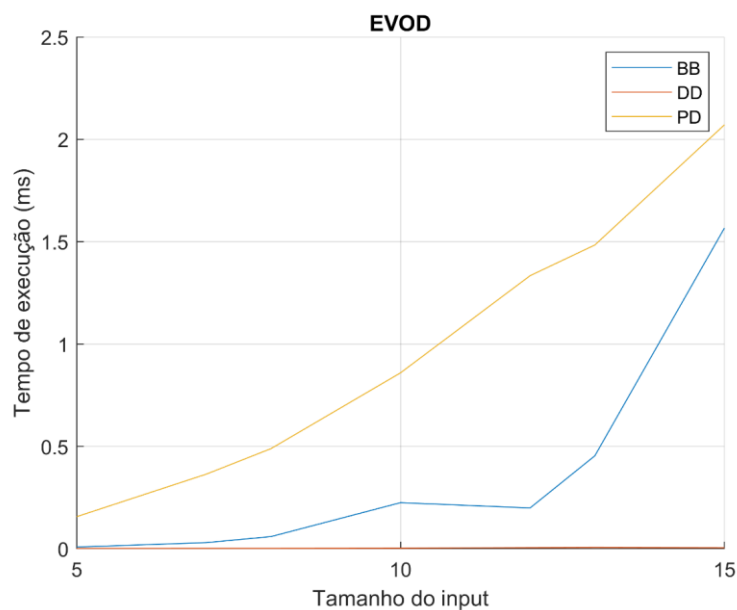


Figura 11. Tempo de execução para a instância Even/Odd

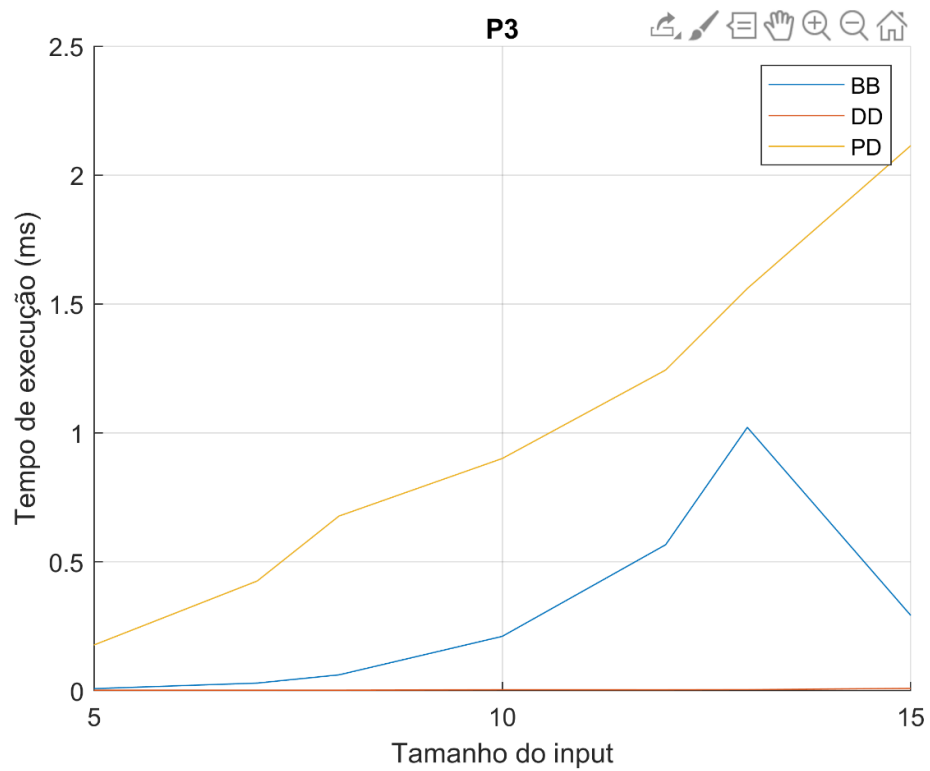


Figura 12. Tempo de execução para a instância P3

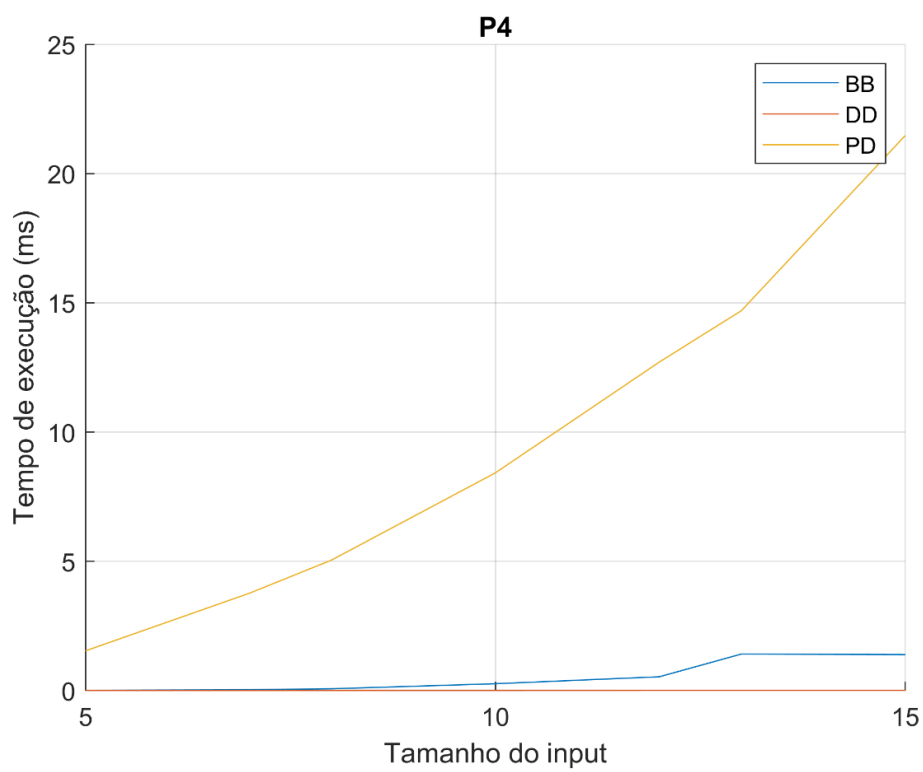


Figura 13. Tempo de execução para a instância P4

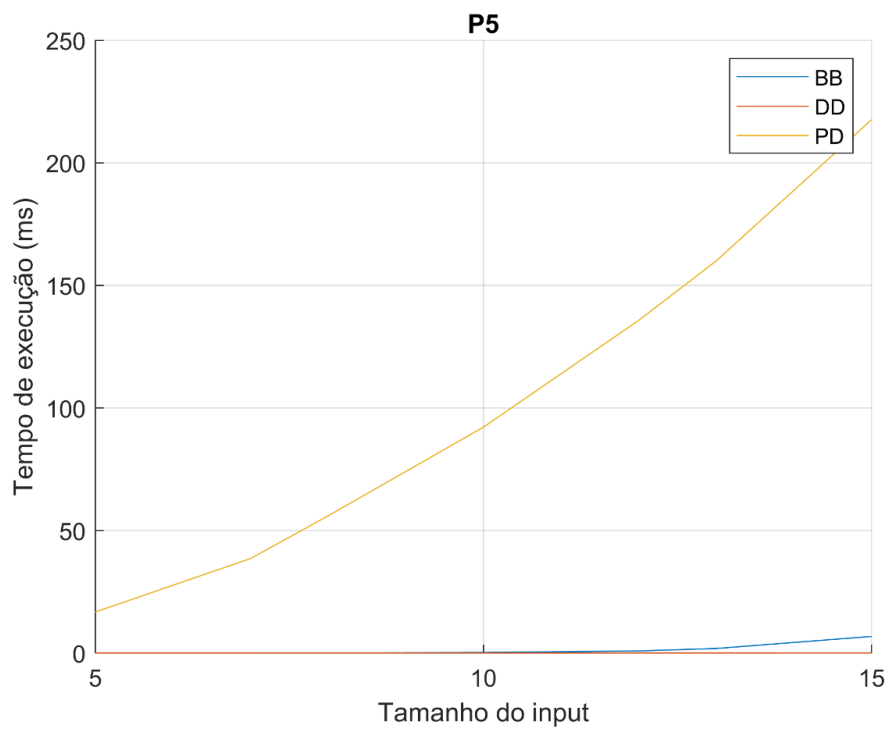


Figura 14. Tempo de execução para a instância P5

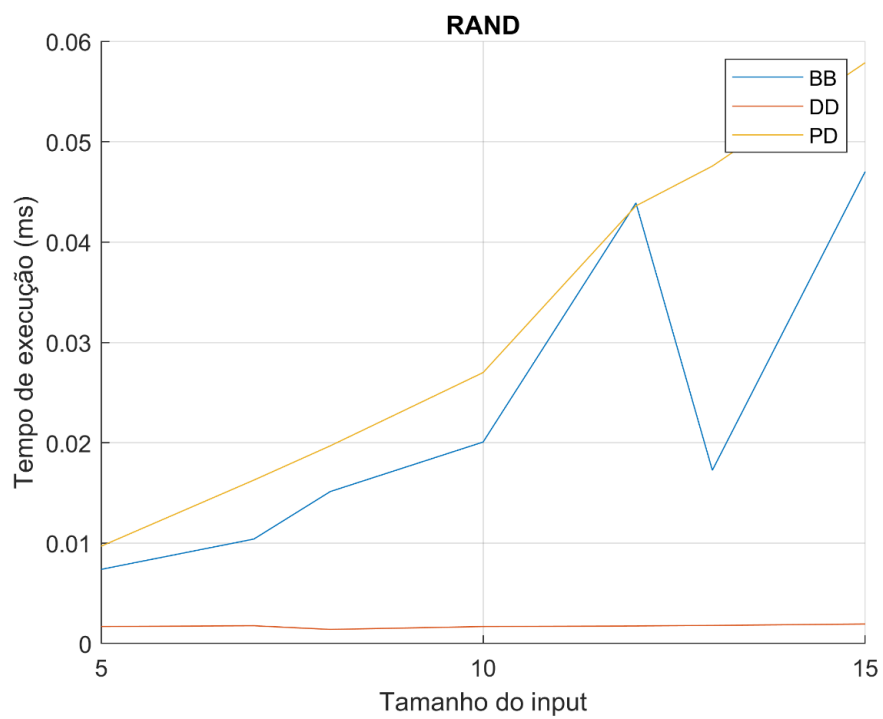


Figura 15. Tempo de execução para a instância Rand

Como esperado, para todas as instâncias o DD (gabarito) foi muito mais rápido, visto que é um método implementado justamente de modo a otimizar o tempo de execução.

Dado que o método PD é $O(W.n)$, sendo W o peso máximo da mochila, espera-se que seu comportamento para valores pequenos do peso, mesmo para valores grandes de entrada (n), seja rápido. Porém, como os pesos crescem bastante dependendo da instância, o PD acaba apresentando um tempo de execução muito superior aos demais.

Além disso, o BB apresenta um melhor desempenho para uma quantidade de entradas maior, mas com uma capacidade da mochila menor. Como no método AVIS a capacidade é dada por $C = \left\lfloor \frac{n-1}{2} \right\rfloor * n * (n-1) + \binom{n}{2}$, tem-se uma capacidade máxima muito elevada quando aumentamos n , o que aumenta bastante o tempo de execução pelo backtracking, que é recursivo.