CES-12 - Relatório Lab 4 Versão 2 - Paradigmas de Programação

Aluno: Bruno Costa Alves Freire

19 de Junho de 2020

1 Problema das Moedas de Troco

Atualização: foi corrigido o exemplo de problema para o qual o paradigma de divisão e conquista performa bem. Em vez de buscar o máximo elemento de um vetor ordenado, o correto seria busca em vetor ordenado.

No problema das moedas de troco, foram implementados 3 algoritmos, cada um seguindo um paradigma de programação, Divisão e Conquista, Programação Dinâmica, e Guloso - ou *Greedy*. Os algoritmos foram testados em problemas pequenos para dois sistemas de denominações: o sistema brasileiro, onde $D = \{1, 5, 10, 25, 50, 100\}$, e um sistema criado especialmente para explorar a sub-otimalidade da estratégia gulosa, onde $D = \{1, 10, 25\}$. A seguir apresentamos comparações de tempo de execução e da solução produzida por cada algoritmo, isto é, a quantidade de moedas encontrada na solução do problema.

1.1 Comparação entre DC, PD e GR

Por razões de praticidade, as comparações envolvendo o algoritmo DC foram realizadas apenas para casos pequenos, uma vez que este algoritmo leva um tempo proibitivamente grande em casos maiores.

Na figura 1, temos a comparação de tempo de execução entre os três algoritmos para valores de troco de até 40, no sistema de denominações brasileiro. O gráfico foi feito em escala logarítmica, para permitir melhor visualização dos tempos de cada algoritmo.

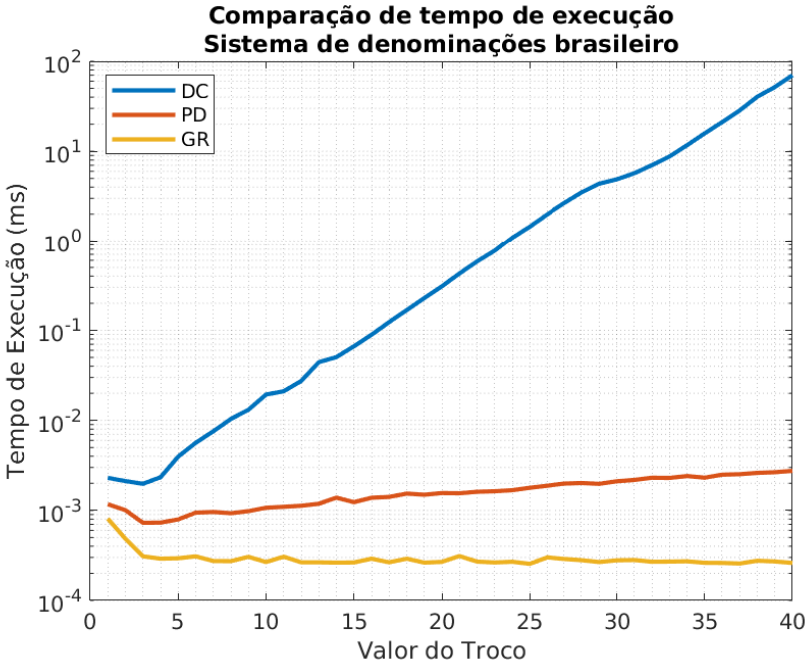

Figura 1: Comparativo de tempo de execução entre os algoritmos DC, PD e GR.

Podemos notar que o algoritmo DC é de fato muitas ordens de grandeza mais lento que os demais. Para termos um vislumbre da execução desse algoritmo, plotamos na figura 2 a quantidade de chamadas recursivas realizadas pelo algoritmo para a resolução do problema, em função do valor do troco.

No eixo esquerdo temos a quantidade de chamadas em escala linear, cuja curva nos sugere um comportamento exponencial. No eixo direito, temos a mesma curva em escala logarítmica, cujo aspecto parece razoavelmente linear, confirmando nossa suspeita.

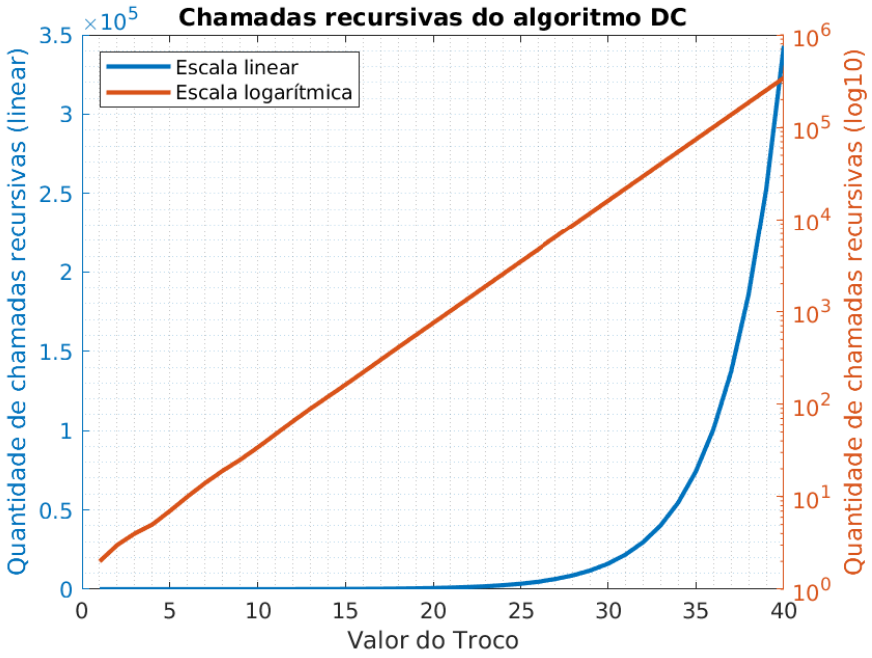

Figura 2: Quantidade de chamadas recursivas do algoritmo de Divisão e Conquista em função do valor do troco.

A seguir, comparamos a resposta de cada algoritmo ao problema do troco, plotando a quantidade de moedas na solução encontrada para o problema em função do valor do troco. O gráfico consta na figura 3. Note que a quantidade de moedas encontrada por cada algoritmo coincide para todos os valores do troco. Isso quer dizer que todos encontraram uma solução ótima (porque sabemos que há garantia de otimalidade para pelo menos um dos algoritmos).

1.2 Comparação aprofundada entre PD e GR

Para nossas próximas análises, vamos executar os algoritmos para valores maiores de troco, o que torna impraticável incluir o algoritmo DC nas comparações.

Para explorar a subotimalidade do algoritmo GR, vamos realizar as comparações com dois sistemas de denominações. Na figura 4 temos a comparação entre os dois algoritmos no sistema de denominações brasileiro. As curvas coloridas são referentes ao eixo dos tempos de execução (em escala logarítmica), enquanto as marcações em preto são relativas ao eixo da quantidade de moedas.

Note que no sistema de denominações brasileiro, ambos os algoritmos obtêm soluções ótimas, embora o algoritmo GR seja mais rápido.

Na figura 5, temos a mesma comparação mas desta vez no sistema de denominações dado por $D = \{1, 10, 25\}$. Desta vez, note que as soluções obtidas pelo algoritmo GR nem sempre são ótimas. Embora o algoritmo continue sendo mais rápido, ele perde otimalidade.

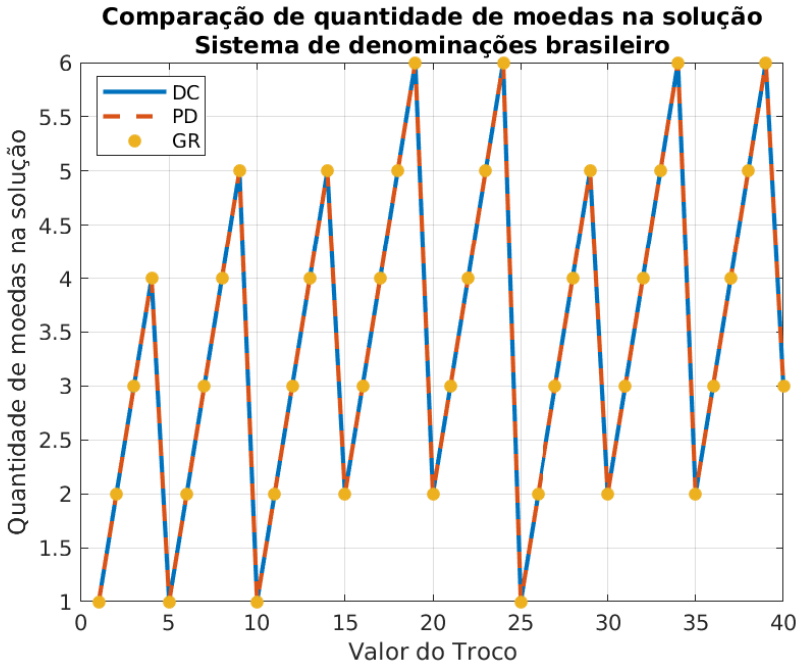

Figura 3: Comparativo da solução encontrada por cada algoritmo em função do valor do troco.

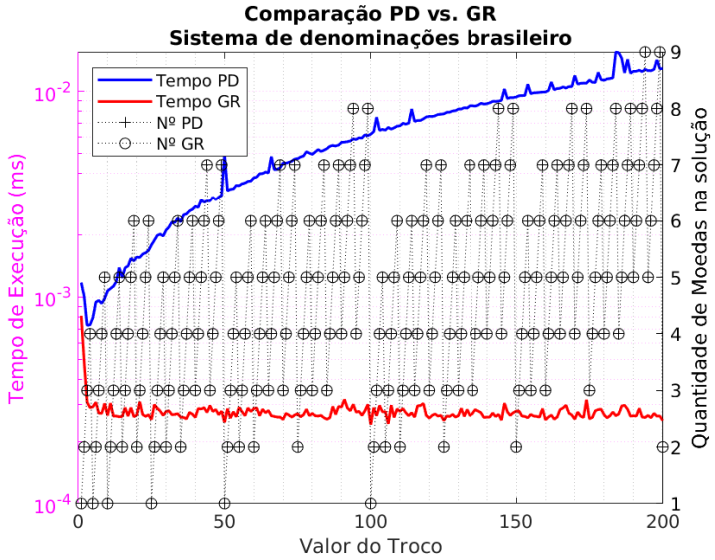

Figura 4: Comparação entre PD e GR em termos de tempo de execução e quantidade de moedas para o sistema de denominações brasileiro.

1.3 Comparação e discussão dos aspectos de tempo e otimalidade

Analisando o crescimento das chamadas recursivas do algoritmo DC, temos a pista de que sua complexidade é exponencial. Isso ocorre porque a estrutura do problema não permite que seja subdividido em problemas menos *disjuntos*, de modo que o algoritmo eventualmente recalcula soluções de um mesmo subproblema várias vezes. Além disso, para armazenar as quantidades de cada denominação usadas na solução, é preciso sobrescrever o vetor com as quantidades toda vez que uma solução melhor é encontrada. Contudo, o algoritmo ainda atinge a otimalidade por dividir o problema de maneira exaustiva em subproblemas menores.

O algoritmo PD é mais rápido pois sua proposta é justamente evitar o retrabalho presente na abordagem DC fazendo uso de armazenamento em memória das soluções dos subproblemas. Assim, troca-se o tempo de executar uma chamada recursiva pela consulta de uma tabela, o que melhora muito o tempo de execução, pois efetivamente reduz a complexidade (de tempo) do algoritmo. Outra

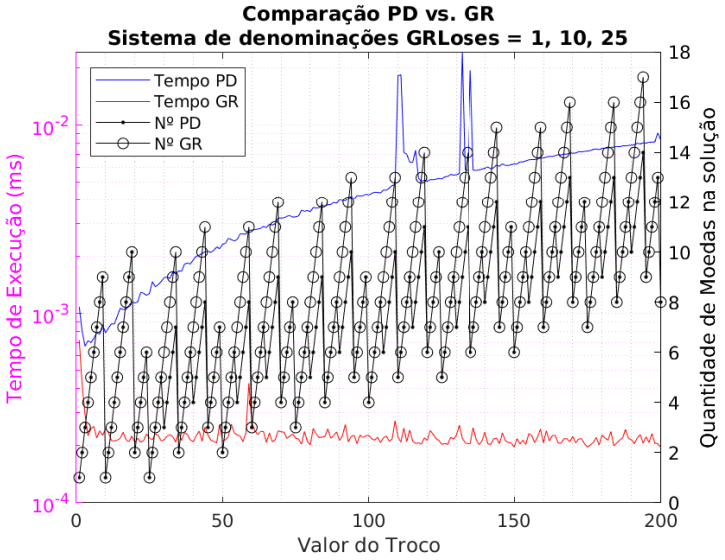

Figura 5: Comparação entre PD e GR em termos de tempo de execução e quantidade de moedas para o sistema de denominações $D = \{1, 10, 25\}$.

diferença que permite um ganho de desempenho, no caso em que precisamos obter as quantidades de cada denominação utilizadas, é que no PD podemos reconstruir a solução apenas após preencher toda a tabela, não sendo necessário sobrescrever o vetor a cada nova solução encontrada. A otimalidade é alcançada pelas mesmas razões que o algoritmo DC, pois a solução é construída sobre as soluções ótimas dos subproblemas.

O algoritmo GR por sua vez é o mais rápido de todos pois a *heurística* empregada é extremamente simples: basta percorrer o vetor de denominações e a cada iteração já teremos a quantidade final de moedas daquela denominação. De fato, se considerarmos que as operações aritméticas são realizadas em $O(1)$, a complexidade da abordagem gulosa é assintoticamente proporcional ao tamanho do vetor de denominações. Ou seja, eliminamos toda a complexidade combinatória oriunda do valor do troco, um fator que influencia o desempenho dos outros algoritmos. Por essa razão, perdemos a otimalidade, pois não há garantias de uma solução ótima do problema completo contenha a solução gulosa dos subproblemas.

1.4 Discussão da estrutura do problema e o desempenho do DC

Na discussão anterior, devemos ressaltar que as garantias de otimalidade para o DC e o PD (e a ausência dessa garantia para o GR) estão fortemente relacionadas à estrutura do problema.

Agora, para entendermos por que o DC apresenta um desempenho ruim nesse problema, devemos destacar que o "espírito" do paradigma de divisão e conquista é resolver subproblemas *disjuntos*. Ou seja, a estrutura do problema deve nos permitir resolver os subproblemas e combinar suas soluções de maneira independente das outras partes, assim, uma vez resolvido um subproblema, toda a informação proveniente dele já foi processada e incorporada à solução. Sob essas condições, o DC performará bem.

Um exemplo simples de problema com uma estrutura favorável ao DC é de busca num vetor ordenado. O procedimento a ser feito é a busca binária, que a cada etapa divide o problema em subproblemas disjuntos, de modo que nunca é necessário retornar a uma metade do vetor após tê-la descartado.

O problema do troco por outro lado não possui essa estrutura, pois ao incorporarmos uma moeda e resolvermos o subproblema para um troco menor, não temos garantia de que a moeda incorporada anteriormente irá realmente fazer parte da solução final.

2 *Subset Sum Problem* - SSP

2.1 Descrição da Implementação

Programação Dinâmica

A solução por programação dinâmica consiste em criar uma tabela de valores lógicos $(n+1) \times (W+1)$, indicando se há solução com os i primeiros itens para o valor j, na entrada ij. A tabela é toda inicializada com `false`, e a primeira coluna com `true`. A partir daí, percorre-se cada linha a partir da coluna de valor w_i, pois para valores menores a existência de solução independe do i-ésimo item, até a última coluna, eventualmente parando na de valor da soma parcial dos itens até o i-ésimo, pois para valores maiores a solução claramente não é possível. Em cada iteração, as entradas da tabela são preenchidas com base nas entradas da linha anterior segundo a recursão de Bellman. Ao final, caso a solução seja encontrada, preenchemos o vetor de saída percorrendo a tabela do final para o início.

Branch & Bound

O algoritmo BB implementado foi uma versão simplificada do método *Branch & Bound*. A implementação consiste em percorrer os itens do maior para o menor, avaliando duas possibilidades: a de resolver o problema *com* aquele item, e a de resolvê-lo *sem* o item, e resolver os subproblemas recursivamente. A cada nível da árvore de recursão estamos analisando um item diferente, sendo a raiz a análise para o item de maior peso, e as folhas analisam a inserção do item de menor peso.

Os casos base (as folhas) são resolvidos simplesmente checando se a soma restante é igual ao menor ou item ou igual a zero. A primeira regra de poda se baseia em checar no início da chamada recursiva se a soma restante é zero, caso que indicaria que uma solução já foi encontrada. A segunda regra de poda é relativa ao teste *com* o item: se o peso do item atual é maior do que a soma requerida, não existe solução com esse item. A terceira regra de poda é relativa ao teste *sem* o item: se a soma parcial dos itens menores que o item atual for menor que a soma requerida, não existe solução sem esse item. A última regra de poda consiste em não abrir a subárvore *sem* o item caso uma solução tenha sido encontrada na subárvore *com* o item.

Uma pequena melhoria implementada consiste em construir o vetor de somas parciais dos pesos dos itens antes de entrar no método recursivo. A partir desse vetor é possível obter o peso de cada item fazendo apenas a diferença entre somas parciais consecutivas. Por outro lado, se fosse necessário computar essas somas parciais a partir do vetor de entradas, seria necessário calcular uma soma de k termos para cada chamada no $(n-k)$-ésimo nível da árvore. O ganho de desempenho torna o algoritmo impressionantes 2 vezes mais rápido, no pior caso.

2.2 Comparação dos Resultados

Os algoritmos implementados foram comparados em termos de tempo de execução médio para algumas instâncias do problema SSP, em função do tamanho do vetor de entrada. Os geradores de instâncias utilizados foram **AVIS**, **EVEN/ODD**, **RANDOM**, **P3**, **P4** e **P5**. Os resultados constam nas figuras 6, 7, 8, 9, 10 e 11, respectivamente, onde os gráficos estão em escala logarítmica, para permitir melhor visualização das curvas.

Nota-se imediatamente que em todos os casos, o PD se mostrou muito mais lento que o BB e o gabarito, estando sempre algumas ordens de grandeza mais lento. Com exceção do gerador de instâncias **AVIS**, o desempenho do BB é quase tão bom quanto o do gabarito.

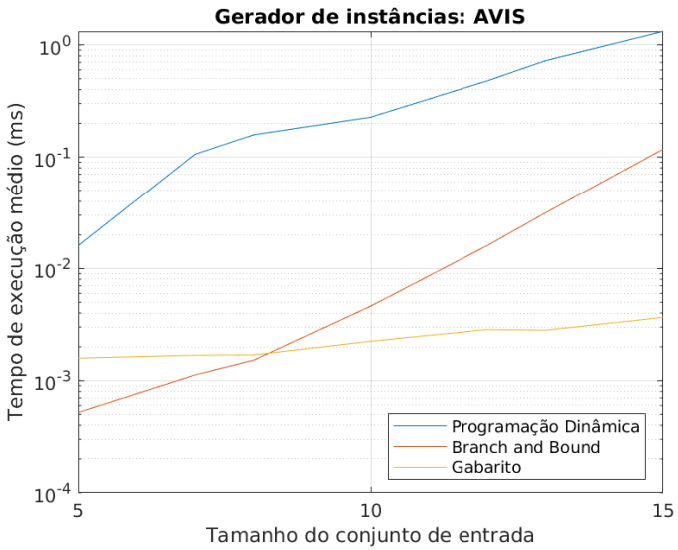

Figura 6: Comparação entre PD e BB em termos de tempo de execução médio nas instâncias do gerador **AVIS**.

O mal desempenho do BB nos problemas **AVIS** se deve ao fato desse gerador criar instâncias que obrigam o algoritmo a explorar a maior parte da árvore possível, e que não têm solução, inutilizando quase sempre as regras de poda. O algoritmo PD por sua vez é prejudicado pelo crescimento de W como $O(n^3)$, tornando sua complexidade final $O(n^4)$.

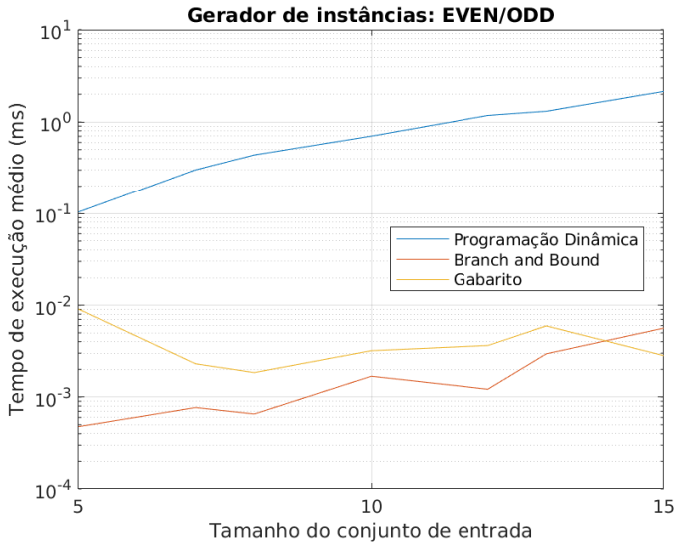

Figura 7: Comparação entre PD e BB em termos de tempo de execução médio nas instâncias do gerador **EVEN/ODD**.

O melhor desempenho de ambos os algoritmos é observado para o gerador de instâncias **RANDOM**, em que temos valores pequenos de W, e os valores das entradas são aleatórios. A limitação de W favorece o PD, pois este tem complexidade $O(nW)$. Por outro lado, a distribuição uniforme dos valores das entradas faz com que, em média, o algoritmo BB se beneficie de suas regras de poda, melhorando seu desempenho. Para o gerador **EVEN/ODD**, o BB é ligeiramente prejudicado por não existir solução nas instâncias, o que impede o algoritmo de aproveitar a regra de poda nº 4. Por outro lado, W agora é fixo, o que beneficia o PD.

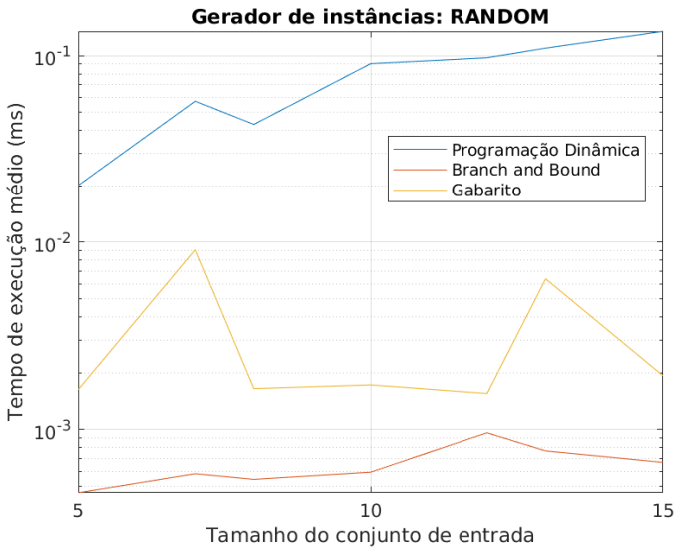

Figura 8: Comparação entre PD e BB em termos de tempo de execução médio nas instâncias do gerador **RANDOM**.

Para os geradores de instâncias PX, temos que W é exponencial em X, e portanto devemos um aumento de uma ordem de grandeza no tempo do PD a cada gerador, o que pode ser constatado comparando-se as escalas nas figuras 9, 10 e 11. O algoritmo BB por sua vez não apresenta variações relevantes de desempenho nesses geradores, uma vez que não é afetado pelo tamanho dos números w_j e W.

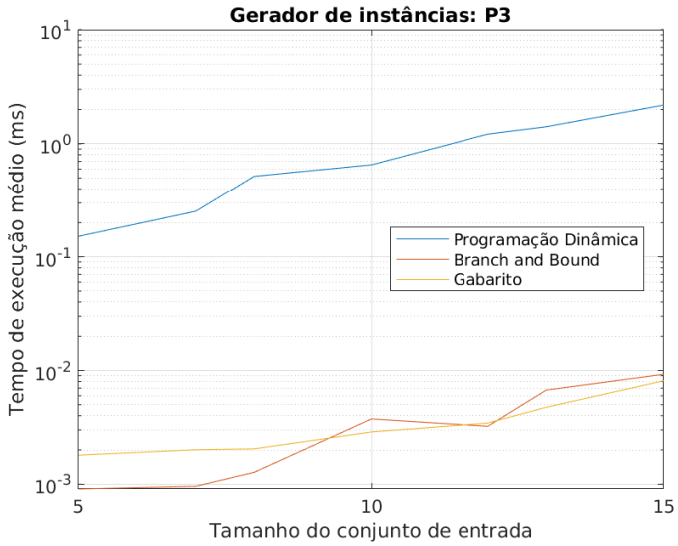

Figura 9: Comparação entre PD e BB em termos de tempo de execução médio nas instâncias do gerador **P3**.

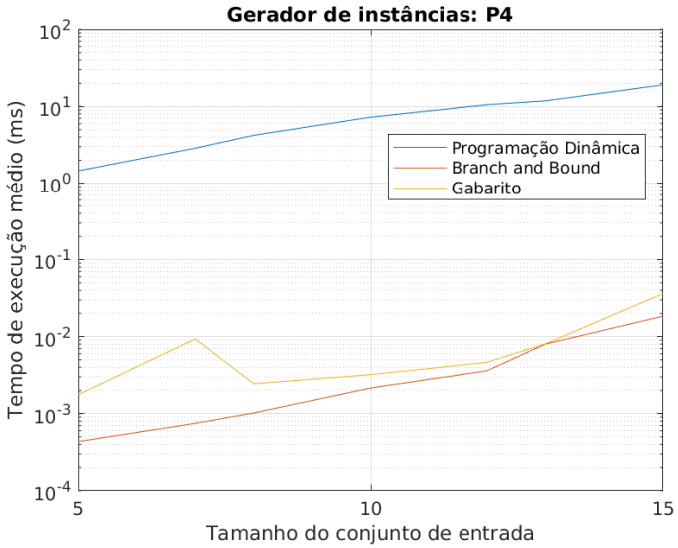

Figura 10: Comparação entre PD e BB em termos de tempo de execução médio nas instâncias do gerador **P4**.

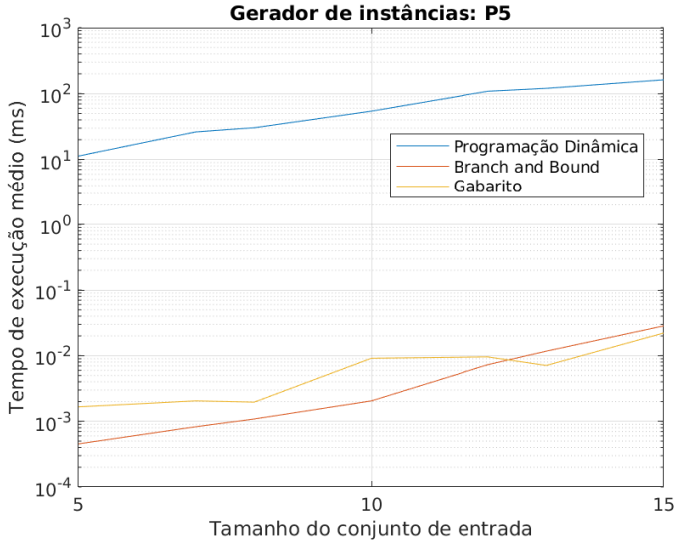

Figura 11: Comparação entre PD e BB em termos de tempo de execução médio nas instâncias do gerador **P5**.