Universidade Federal do ABC

Centro de Matemática, Computação e Cognição

Relatório Final PDPD – Edital 11/2022

A Combinatória do Problema do Troco

Aluno:

Enzo Patryck Teixeira Gonçalves

Orientador:

Carla Negri Lintzmayer

Santo André – SP

2023

Resumo

Este projeto de iniciação científica teve como tema a combinatória do Problema do Troco, um problema clássico em que, dada uma quantidade finita de valores nominais de moeda e um valor total n, deseja-se encontrar a menor quantidade de moedas cuja soma seja n. Por exemplo, se os valores das moedas são $1, 5, 10, 25$ e 50 e $n = 137$, então o menor número de moedas é 6 $(50 + 50 + 25 + 10 + 1 + 1)$. Se os valores das moedas são $1, 5, 9$ e 16 e $n = 18$, então o menor número é 2 $(9 + 9)$.

O estudo incluiu, inicialmente, a análise de um algoritmo guloso para o problema e uma prova de corretude para o mesmo. Tal algoritmo oferece a resposta cuja escolha é pela moeda com o maior valor possível. Esse não é o caso do segundo conjunto de moedas acima, cuja escolha gulosa nos daria 3 moedas $(16 + 1 + 1)$, mas é o caso do primeiro conjunto. Conjuntos de moedas cuja solução dada pelo algoritmo guloso é ótima são chamados de *canônicos*.

Em seguida estudamos resultados que incluem o estudo da caracterização de alguns conjuntos canônicos por meio de levantamentos bibliográficos. Além disso, a pesquisa também abrangeu o estudo do algoritmo de programação dinâmica, que fornece uma solução ótima independentemente do conjunto de moedas e leva tempo pseudo-polinomial. De fato, esse problema é um caso especial do clássico Problema da Mochila e, portanto, NP-difícil.

O objetivo desse material é fornecer uma base sólida para futuros estudantes interessados no Problema do Troco, tornando-se um material didático relevante para os que se dedicam a essa área.

Palavras-chave: Problema do Troco; Algoritmo Guloso; Conjuntos Canônicos.

Sumário

1 Introdução

No Problema do Troco nos é dada uma quantidade finita de valores nominais de moeda e um valor total n e deseja-se encontrar a menor quantidade de moedas cuja soma seja n. Por exemplo, se os valores das moedas são 1, 5, 10, 25 e 50 e $n = 137$, então o menor número de moedas é 6 $(50 + 50 + 25 + 10 + 1 + 1)$. Se os valores das moedas são $1, 4, 8$ e 14 e $n = 17$, então o menor número é 3 $(8 + 8 + 1)$.

Em alguns cenários, é possível encontrar uma solução eficiente para o Problema do Troco utilizando a estratégia conhecida como *método guloso*. Essa abordagem envolve o uso repetido da moeda de maior valor disponível, na maior quantidade de vezes possível, antes de passar para a próxima moeda de valor mais alto, e assim sucessivamente, até alcançar a solução desejada. Essa estratégia devolve solução ótima para o primeiro exemplo dado acima, mas não para o segundo. Utilizando a estratégia gulosa para os valores de moedas $1, 4, 8$ e 14 e $n = 17$, descobriremos que a quantidade de moedas utilizadas é 4 $(14 + 1 + 1 + 1)$. No entanto, isso não corresponde à solução ótima, que seria composta por apenas 3 moedas $(8 + 8 + 1)$. Esse exemplo ilustra uma das limitações da estratégia gulosa: ela pode não ser a abordagem exata para todos os conjuntos de moedas e valores de troco.

Portanto, o estudo do Problema do Troco não se limita apenas a encontrar soluções, mas também a compreender as complexidades inerentes a diferentes conjuntos de moedas e a busca por métodos que garantam a otimalidade em uma variedade de cenários. Além disso, ele atua como uma ferramenta de ensino valiosa, ilustrando conceitos complexos de algoritmos de maneira acessível, ao mesmo tempo, em que serve como base essencial para estratégias de otimização mais avançadas. A análise das situações em que a estratégia gulosa não fornece soluções ótimas é fundamental para desenvolver abordagens mais sofisticadas quando necessário.

O objetivo desse material é fornecer uma base sólida para futuros estudantes interessados no Problema do Troco, tornando-se um material didático relevante para os que se dedicam a essa área.

Ao longo das seções desse texto, exploraremos conceitos fundamentais para a compreensão da análise de algoritmos. Começaremos, na Seção 2, definindo algoritmos e

utilizando pseudo-código como uma representação genérica para os mesmos. Enfatizaremos a importância da corretude, destacando a prova por invariante de laço. Em seguida, abordaremos a análise do tempo de execução e a noção de otimalidade, com um foco especial no contexto do Problema do Troco. Na Seção 3 introduziremos o conceito de algoritmo guloso e exploraremos os conceitos de conjuntos canônicos de moedas, fornecendo uma base sólida para a compreensão dessas estruturas. Na Seção 4 apresentaremos uma série de teoremas que tratam do Problema do Troco e Conjuntos Canônicos. Por fim, na Seção 5 faremos uma breve menção à programação dinâmica como uma abordagem alternativa para a resolução do Problema do Troco, enquanto que na Seção 6 apresentamos algumas observações finais.

2 Fundamentação Teórica

Nessa seção apresentaremos conceitos básicos importantes relacionados com algoritmos.

2.1 Algoritmos

Algoritmos são sequências de instruções que recebem dados de entrada e geram uma solução para um determinado problema. Problemas computacionais, por exemplo, podem ser resolvidos por meio de algoritmos que, a partir dos dados de entrada, calculam uma solução viável com base nas instruções previamente definidas. Para ser considerado correto, um algoritmo precisa encontrar soluções para qualquer instância do problema, ou seja, deve haver uma solução para qualquer conjunto de valores específicos da entrada. Se um algoritmo não devolver uma solução para pelo menos uma instância do problema, ele não é considerado correto [3].

2.2 Pseudo-código

Para representar um algoritmo, utilizaremos pseudo-código, que é uma forma genérica de escrever um algoritmo. Não será utilizada nenhuma linguagem de programação específica, a fim de tornar o conteúdo acessível a um público amplo. Veja no Algoritmo 1 um exemplo de pseudo-código para um algoritmo de BuscaLinear, que recebe um vetor A contendo n

elementos e uma chave de busca k, e devolve um valor i, onde $1 \leq i \leq n$, caso um elemento igual a k seja encontrado. Por outro lado, se tal elemento não for encontrado, o algoritmo devolve -1.

Algoritmo 1 BUSCALINEAR(A, n, k)

```
1: i = 1
2: Enquanto i ≤ n e A[i] ≠ k faça
3:     i = i + 1
4: Se i ≤ n e A[i] == k então
5:     Devolve i
6: Devolve −1
```

O algoritmo BUSCALINEAR opera de forma bastante simples. Inicialmente, definimos a variável i como 1, indicando a posição inicial no vetor A a ser analisada. Para progredir, incrementamos o valor de i em uma unidade enquanto duas condições no laço enquanto forem atendidas: $i \leq n$ e $A[i] \neq k$. Quando o valor de k é encontrado, o laço enquanto é encerrado e o algoritmo devolve o índice i tal que $A[i] = k$. No entanto, se k não for encontrado, o algoritmo devolve -1, indicando que nenhum elemento com a chave k foi localizado no vetor A.

2.3 Corretude

Ao testar o Algoritmo 1 com diferentes valores de entrada é possível averiguar que ele funciona e retorna a resposta correta, mas é preciso ter certeza que isso se aplica para qualquer entrada.

Para provar que um algoritmo não funciona é simples, pois é preciso somente apresentar um caso de entrada onde o algoritmo não devolva a resposta correta. Mas, para mostrar que ele é correto não é tão direto. Como os possíveis valores de entrada são infinitos, é impossível testar todos os casos, portanto precisamos de uma maneira viável para provar a corretude do algoritmo. Para isso pode ser usada uma invariante de laço (laços **para** ou **enquanto**).

A **invariante de laço** é uma propriedade P do algoritmo verdadeira no início de cada iteração do laço, para qualquer valor da variável que está sendo iterada. Para provar a invariante de laço, é utilizada uma prova por indução na quantidade de vezes que o teste do laço executa. É provado o caso base, $P(1)$, assume-se que $P(t)$ é verdadeira para $t \geq 1$

e, em seguida, é provado que $P(t+1)$ também é verdadeira.

Queremos estabelecer uma invariante de laço que sirva para demonstrar a correção do algoritmo. O que é feito durante cada iteração do laço só terá impacto quando todas as iterações terminarem. Assim, seja T a quantidade total de vezes que o laço executa. Note que $P(T)$ fornece informações sobre variáveis importantes no início da última iteração, mas essas variáveis mudarão ao longo dessa iteração, tornando $P(T)$ obsoleta. O que realmente importa é $P(T+1)$, pois é verdadeira antes da $(T+1)-$ésima iteração, que não vai acontecer e, portanto, se mantém válida ao final da execução do laço.

Considere novamente o Algoritmo 1. Na BuscaLinear, nosso objetivo é estabelecer que "um elemento igual a k existe no vetor **se e somente se** o algoritmo retorna o índice que contém tal elemento". Essa afirmação pode ser desdobrada em duas partes: primeiro, "se um elemento igual a k existe, então o algoritmo devolve seu índice (que é um valor entre 1 e n)"; segundo, "se o algoritmo devolve um índice entre 1 e n, então isso significa que um elemento com a chave k existe". A última afirmação é equivalente, por contrapositiva, a dizer que se um elemento com essa chave não existe, então o algoritmo devolverá um índice que não está dentro do intervalo de 1 a n, ou seja, -1.

Para demonstrar essa proposição, é necessário introduzir uma invariante de laço.

Invariante: Laço enquanto – BuscaLinear

$P(t)$ = "Antes da t-ésima iteração começar, vale que $i = t$ e o vetor $A[1 \ldots i-1]$ não contém um elemento igual a k."

Demonstração. Vamos agora realizar uma prova por indução no número t de vezes que o teste de parada é executado.

Caso base: Quando $t = 1$, o teste executou uma vez. Temos $i = 1$, e como $A[1..0]$ é vazio, por vacuidade, não contém um elemento com a chave k. Portanto, $P(1)$ é verdadeira.

Passo: Suponhamos que o teste de parada foi executado t vezes, com $t \geq 1$. Isso indica que a t-ésima iteração está prestes a começar. Assumindo que $P(t)$ seja verdadeira, precisamos mostrar que $P(t+1)$ também é. Como $P(t)$ é verdadeira, sabemos que no início da t-ésima iteração, $i = t$ e $A[1..t-1]$ não contém um elemento com a chave k.

Devido ao resultado do t-ésimo teste, sabemos que $A[t] \neq k$. Portanto, ao final

da t-ésima iteração, temos que $A[1..t]$ não contém um elemento com a chave k. Isso valida $P(t+1)$, pois i é atualizado para $t+1$.

Concluímos, portanto, que P é uma invariante de laço. □

Demonstraremos a corretude do algoritmo utilizando a invariante estabelecida. O término do laço ocorre quando: (1) $i > n$ ou (2) $i \leq n$ e $A[i] = k$. Se $i > n$, então $P(n+1)$ garante que o vetor $A[1..n]$ não contém um elemento igual a k. De fato, quando $i > n$, o algoritmo devolve -1, que é o esperado. Se o laço acabou porque $i \leq n$ e $A[i] = k$, então é porque existe um elemento igual a k em A e, de fato, o algoritmo devolve i nesse caso.

É importante destacar que essa prova não faz suposições sobre os valores das chaves dos elementos de A ou sobre o valor de k. Portanto, podemos concluir que o algoritmo funciona corretamente para qualquer instância.

2.4 Tempo de Execução

Além de provar que o algoritmo funciona, também estamos interessados em sua eficiência, ou seja, quanto tempo ele leva para ser executado. Muitos fatores alteram esse tempo, como a velocidade do processador, a linguagem de programação utilizada, o sistema operacional em uso e o tamanho da entrada. Por isso, para determinar a velocidade de um algoritmo, é utilizado um modelo de computação mais simples que conta os passos básicos que podem ser realizados rapidamente em números pequenos (que podem ser escritos em até 64 *bits*). Esses passos básicos incluem operações aritméticas, relacionais e lógicas, movimentações em variáveis simples e operações de controle. O tempo de execução é escrito como uma função que depende do tamanho da entrada.

Geralmente o tamanho da entrada será dada por um ou mais números naturais. Portanto, ao denotar o tamanho de uma entrada como n, será utilizada a função $T(n)$ para indicar quantos passos básicos o algoritmo realizará. No caso de ter mais de um parâmetro que definem o tamanho da entrada, como n e m, utilizamos a função $T(n, m)$, e assim sucessivamente.

Vamos examinar o Algoritmo 2, que apresenta a função SOMATORIO, o qual recebe um vetor A contendo n números e se propõe a calcular a soma dos valores armazenados em A. Em outras palavras, sua função é determinar o resultado de $\sum_{j=1}^{n} A[j]$.

Algoritmo 2 SOMATORIO(A, n)

1: $soma = 0$
2: **Para** $i = 1$ até n, incrementando **faça**
3: $\quad soma = soma + A[i]$
4: **Devolve** *soma*

Neste algoritmo de soma de um vetor de tamanho n, cada iteração consiste em quatro atribuições ($soma = 0$, $i = 1$, $soma = soma + A[i]$, $i = i + 1$), um teste lógico ($i \leq n$), duas operações aritméticas ($i + 1$, $soma + A[i]$), um acesso ao vetor ($A[i]$), e um retorno. Cada operação leva um tempo t. O corpo do laço é executado n vezes, levando $3tn$ de tempo, enquanto o teste do laço ocorre $n + 1$ vezes.

Resumindo, o tempo $T(n)$ de execução de SOMATORIO é:

$$T(n) = \underbrace{t}_{soma=0} + \underbrace{t}_{i=1} + \underbrace{t(n+1)}_{i \leq n} + \underbrace{2tn}_{i=i+1} + \underbrace{3tn}_{soma=soma+A[i]} + \underbrace{t}_{\textbf{devolve}\ \text{soma}} = 6tn + 4t.$$

Nesse algoritmo, note que independente dos valores de A, o tempo de execução será sempre o mesmo. No entanto, em algoritmos mais complexos o tempo pode variar de acordo com o conteúdo da entrada. Por exemplo, considere o algoritmo BUSCALINEAR novamente.

Vamos considerar inicialmente que há um elemento com chave k no vetor $A[1..n]$, onde p_k é a sua posição em A ou seja, $A[p_k] = k$. A linha 1 é executada uma vez (1 *operação*), as linhas 4 e 5 são executadas uma vez (6 *operações*). Os testes do laço na linha 2 são executados p_k vezes (5 *operações*), e a linha 3 é executada $p_k - 1$ vezes (2 *operações*). Portanto, o tempo total de execução $T_{BLE}(n)$ de BUSCALINEAR(A, n, k) quando um elemento com chave k está em A é:

$$T_{BLE}(n) = t + 5tp_k + 2t(p_k - 1) + 6t = 7tp_k + 5t.$$

Assim vemos que o tempo de execução da BUSCALINEAR depende da posição do elemento buscado no vetor. Em caso de sucesso, se o elemento está na última posição de A, o tempo de execução é de $7tn + 5t$, enquanto se estiver na primeira posição, o tempo é de $12t$. E quando o elemento não está presente em A, o tempo de execução é $7tn + 8t$.

Podemos afirmar que o tempo de execução $TBL(n)$ de BUSCALINEAR(A, n, k) para

qualquer entrada satisfaz:

$$12t \leq T_{BLE}(n) \leq 7tn + 8t.$$

A análise revela que, para instâncias de tamanho n, o tempo de execução varia devido à posição de k no vetor, resultando em diferentes cenários de melhor e pior caso. Essa análise do tempo de execução é crucial para entender o desempenho do algoritmo em diversas situações.

2.5 Notação Assintótica

A análise assintótica é uma abstração que ajuda a entender o tempo de execução de algoritmos, utilizando taxas de crescimento de funções descritas pelo tempo de execução em passos básicos. Ela permite comparar os tempos de execução independentemente de fatores físicos citados anteriormente, como a velocidade da máquina ou sistema operacional.

No estudo da eficiência dos algoritmos, não concentramos nossa atenção nas constantes específicas, mas sim nas características gerais das funções que descrevem o tempo de execução. Para ilustrar, se um algoritmo consome tempo segundo a função $f(n) = an^2 + bn + c$, onde a, b e c são constantes, e n é o tamanho da entrada, o que realmente importa, para valores grandes de n, é o n^2.

Por exemplo, considere as funções $f(n) = 475n + 34$ e $g(n) = n^2 + 3$. É evidente que $f(n) \leq g(n)$ não é sempre verdade, já que, por exemplo, $f(4) = 1934$ e $g(4) = 19$. No entanto, à medida que n cresce além de 476, sempre teremos $f(n) \leq g(n)$. Além disso, podemos notar que $f(n) \leq 475g(n)$ para qualquer $n \geq 1$, uma vez que $f(n) = 475n+34 \leq 475n^2 + 1425 = 475g(n)$. Nesse contexto, as constantes que multiplicam as funções e os valores iniciais de n são ofuscados pela notação assintótica, concentrando nossa atenção nas tendências de crescimento à medida que n aumenta.

As notações assintóticas são formalismos que oferecem maneiras de concentrar a atenção no crescimento de funções, permitindo-nos ignorar detalhes que se tornam insignificantes à medida que o tamanho da entrada aumenta. Isso é crucial porque, para valores pequenos de n, as características das funções podem variar significativamente.

Entre essas notações, a mais comumente usada é a notação "O" (*big O*), que fornece uma cota superior assintótica para o tempo ou espaço de um algoritmo em termos do

tamanho da entrada. Essa notação é valiosa para simplificar a análise, concentrando-se nas partes dominantes do desempenho do algoritmo. Embora existam outras notações assintóticas, como Omega (Ω) e Theta (Θ), que fornecem informações mais detalhadas sobre o desempenho em casos melhores e médios, é mais comum que seja usada apenas a notação O [3].

Por exemplo, podemos dizer que os algoritmos SOMATORIO e BUSCALINEAR levam tempo $O(n)$ em sua execução.

2.6 Otimalidade

Retornando à discussão sobre o Problema do Troco, adentramos em uma consideração importante: a otimalidade dos algoritmos. Até agora, nossos esforços estiveram concentrados em determinar se um algoritmo é funcional e quantificar seu tempo de execução. No entanto, neste ponto, nossa atenção se volta para a qualidade da resposta que ele produz.

Em problemas de otimização, recebe-se algumas restrições como entrada, o que faz com que diferentes respostas possam ser consideradas **viáveis**. Além disso, recebe-se uma função objetivo, que qualifica essas soluções, de forma que deseja-se encontrar uma que minimize ou maximize (otimize) essa função. Uma solução viável que satisfaça isso é chamada de **ótima**. Um algoritmo para um problema de otimização é dito ótimo se ele consegue encontrar uma solução ótima para qualquer entrada.

É importante notar que, nos casos dos algoritmos BUSCALINEAR e SOMATORIO mencionados anteriormente, o conceito de otimalidade não se aplica diretamente. Isso ocorre porque a resposta desejada é única; assim, não há espaço para discussão sobre otimalidade nessas situações, uma vez que a resposta é binária: correta ou incorreta.

Contudo, o Problema do Troco é um problema de minimização e, portanto, tem diferentes soluções viáveis, sendo que nem todas serão ótimas.

2.7 Problema do Troco

Consideraremos as seguintes notações.

Notação:

$M = (c_1, c_2, \ldots, c_k)$ é vetor de moedas — $k =$ Tamanho do vetor $Moedas$

$n =$ valor do troco — $S = (s_1, s_2, \ldots, s_k)$ é vetor de solução

$c_i =$ valor da i-ésima moeda — $s_i =$ quantidade de c_i utilizadas em S

Consideraremos, ainda, que $c_1 > c_2 > \cdots > c_k$ e sempre iremos assumir que $c_k = 1$, garantindo que qualquer valor inteiro de troco possa ser representado.

Chamamos $S = (s_1, s_2, \ldots, s_k)$ de solução viável para o Problema do Troco se ela satisfaz a seguinte expressão:

$$\sum_{i=1}^{k} c_i \times s_i = n \tag{1}$$

Se a representação satisfaz $\sum_{i=j+1}^{k} c_i \times s_i < c_j$, para $1 \leq j < k$, então ela é uma representação gulosa do troco do valor n. Como só é possível construir uma solução viável gulosa, vamos denotá-la por $S(n)$ e denotar por $q(S(n)) = \sum_{i=1}^{k} s_i$ a quantidade de moedas que foram utilizadas para realizar o troco [2].

De forma similar, vamos chamar de $S^*(n) = (s_1^*, s_2^*, \ldots, s_k^*)$ uma solução ótima para o troco do valor n. Da mesma forma, $q(S^*(n)) = \sum_{i=1}^{k} s_i^*$ será a quantidade de moedas utilizadas por essa solução ótima.

3 Algoritmo Guloso para o Problema do Troco

Um algoritmo é chamado guloso quando ele resolve problemas escolhendo sempre a melhor opção disponível no momento, sem analisar as outras opções, mesmo se no futuro elas guiem para uma solução não ótima. A estratégia gulosa para o Problema do Troco já foi citada anteriormente, e nas seções a seguir vamos apresentá-la formalmente.

3.1 Pseudo-código

O algoritmo TrocoGuloso, apresentado no Algoritmo 3, foi projetado para resolver o problema de encontrar a combinação mínima de moedas para representar uma determinada quantia de troco. A seguir estão os detalhes do seu funcionamento.

Inicializa a variável *resto* com o valor do troco a ser calculado. O valor de *resto* será atualizado à medida que as moedas são usadas para compor o troco. Após, se inicia um

Algoritmo 3 TrocoGuloso(n, M, k)

1: $resto = n$
2: seja $S[1..k]$ um vetor
3: **Para** $i = 1$ até k **faça**
4: $\quad piso = \lfloor resto \ / \ M[i] \rfloor$
5: $\quad resto = resto \ \% \ M[i]$
6: $\quad S[i] = piso$
7: **Devolve** S

laço **para** que percorre as moedas disponíveis. O contador i varia de 1 a k.

Para cada tipo de moeda, calcula o *piso* da divisão entre o valor atual de *resto* e o valor da moeda atual $M[i]$. O *piso* é a maior quantidade de moedas desse tipo que pode ser usada para compor o troco sem exceder o valor de *resto*.

O valor de *resto* é atualizado calculando o resto da divisão entre o valor atual de *resto* e o valor da moeda atual $M[i]$. Isso representa o valor restante a ser calculado para o troco após a utilização das moedas do tipo atual.

No fim da iteração atual, é armazenada a quantidade calculada de moedas do tipo atual no vetor S de solução na posição correspondente, $S[i]$.

O laço **para** continua até que todos os tipos de moedas tenham sido considerados.

Por fim, o algoritmo devolve o vetor S como resultado, representando a quantidade de moedas de cada tipo utilizada para compor o troco.

Agora que compreendemos detalhadamente como o algoritmo TrocoGuloso opera, é fundamental estabelecer sua validade e corretude. Vamos adentrar na prova de que este algoritmo, de fato, produz uma solução viável para o Problema do Troco. Levando em consideração que, devido à sua natureza gulosa, ele nem sempre resultará na quantidade mínima absoluta de moedas.

3.2 Prova de Corretude

Para provar a corretude do algoritmo, consideramos a seguinte igualdade como nosso objetivo:

$$\sum_{i=1}^{k} M[i] \times S[i] = n \tag{2}$$

Demonstração. Para provar o resultado, precisamos da seguinte invariante de laço.

Invariante: Laço para – TrocoGuloso

$P(t)$ = "Antes da t-ésima iteração começar, vale que $i = t$, $resto = resto_t$ e $\sum_{j=1}^{t-1} M[j] \cdot S[j] = n - resto_t$".

Case base: $t = 1$. O teste executou uma vez, temos que $i = 1$, $resto = resto_1 = n$, e $\sum_{j=1}^{0} M[j] \times S[j] = 0$, que é $n - resto_{t+1}$. Portanto, o caso base é válido.

Passo: Seja $t \geq 1$ e suponha que $P(t)$ vale. Queremos provar $P(t+1)$: "Antes da $(t+1)$-ésima iteração começar, vale que $i = t+1$, $resto = resto_{t+1}$ e $\sum_{j=1}^{t} M[j] \cdot S[j] = n - resto_{t+1}$".

Na iteração atual $i = t$, a variável *piso* recebe o valor $\lfloor resto_t / M[t] \rfloor$, a variável resto é atualizada para $resto_t \% M[t]$, que é $resto_{t+1}$, e o elemento $S[t]$ recebe o valor de *piso*.

Sabemos que $\sum_{j=1}^{t-1} M[j] \cdot S[j] = n - resto_t$, então $\sum_{j=1}^{t-1} M[j] \cdot S[j]$ somado com a próxima iteração $(M[t] \cdot S[t])$ resulta em $\sum_{j=1}^{t} M[j] \cdot S[j]$. Ou seja, $\sum_{j=1}^{t} M[j] \cdot S[j] = (\sum_{j=1}^{t-1} M[j] \cdot S[j]) + M[t] \cdot S[t]$

Por $P(t)$ temos que $\sum_{j=1}^{t} M[j] \cdot S[j] = n - resto_t + \lfloor resto_t / (M[t] \cdot S[t]) \rfloor$. Simplificando a expressão e aplicando a propriedade distributiva da subtração, obtemos $\sum_{j=1}^{t} M[j] \cdot S[j] = n - (resto_t - \lfloor resto_t / (M[t] \cdot S[t]) \rfloor)$.

Aqui temos uma representação da operação módulo, uma vez que $A \% B = A - \lfloor A/B \rfloor \cdot B$. Substituindo, temos

$$\sum_{j=1}^{t} M[j] \cdot S[j] = n - resto_t \% M[t]\,,$$

onde $resto_t \% M[t]$ é um valor conhecido e pode ser escrito como $resto_{t+1}$. Substituindo na fórmula, temos

$$\sum_{j=1}^{t} M[j] \cdot S[j] = n - resto_{t+1}\,.$$

O laço ainda incrementa i, fazendo $i = t+1$, e acabamos de provar, portanto, $P(t+1)$. Logo, P é invariante. □

3.3 Complexidade de Tempo

O laço do algoritmo TrocoGuloso itera por todas as moedas disponíveis no vetor M e, para cada tipo de moeda, realiza um conjunto fixo de operações, independentemente do tamanho da entrada. Portanto, o tempo de execução desse algoritmo é principalmente determinado pelo número de tipos de moedas disponíveis, denotado por k.

Neste algoritmo cada iteração consiste em quatro atribuições ($resto = n$, $piso = \lfloor resto/M[i] \rfloor$, $resto = resto\%M[i]$ e $S[i] = piso$), duas operações aritméticas ($\lfloor resto/M[i] \rfloor$, um teste lógico ($i \leq k$) e $resto\%M[i]$), dois acessos ao vetor ($A[i]$), e um retorno.

Cada operação leva um tempo t. O corpo do laço é executado k vezes, levando $8tk$ de tempo, enquanto o teste do laço ocorre $k + 1$ vezes. Resumindo, o tempo $T(k)$ de execução de TrocoGuloso é:

$$T(k) = \underbrace{t}_{resto=n} + \underbrace{t}_{i=1} + \underbrace{t(k+1)}_{i \leq k} + \underbrace{2tk}_{i=i+1} + \underbrace{8tk}_{\text{Conteúdo do laço}} + \underbrace{t}_{\textbf{devolve } S} = 11tk + 4t.$$

Para ser mais preciso, podemos expressar a complexidade de tempo do algoritmo como $O(k)$. Isso significa que, à medida que aumentamos o número de tipos de moedas, o tempo de execução do algoritmo aumentará linearmente.

3.4 Conjuntos Canônicos

Os **conjuntos canônicos** são definidos como conjuntos de moedas que possuem uma característica única: a solução encontrada pelo algoritmo guloso usando tais conjuntos sempre será ótima.

Definição 3.1. *Um conjunto de moedas M é canônico se $q(S(n)) = q(S^*(n))$ para qualquer n.*

Um exemplo é o conjunto de moedas utilizado cotação brasileira (*real*) onde $M = (50, 25, 10, 5, 1)$. Este conjunto é canônico segundo a definição acima e qualquer valor de troco pode ser facilmente representada de forma ótima utilizando a estratégia gulosa.

Definição 3.2. *Um conjunto de moedas M não é canônico se existe um valor de troco n onde $q(S(n)) > q(S^*(n))$. Tal n é chamado de contraexemplo de M [6].*

Teorema 3.1 ([2]). *Se um conjunto de moedas M é canônico, então o algoritmo guloso, utilizando esse conjunto, sempre devolve uma solução ótima para qualquer valor de troco n.*

A seguir, apresentamos uma demonstração do Teorema 3.1 para quando o conjunto canônico é $M = (50, 25, 10, 5, 1)$.

Demonstração. Suponhamos um conjunto canônico M e a busca por uma solução para um valor de troco n. Se o algoritmo guloso não produz uma solução ótima, então seja i tal que c_i é a maior moeda para a qual o algoritmo escolheu uma quantidade diferente da solução ótima, isto é, $S(n)[i] \neq S^*(n)[i]$. Mas caso $S(n)[i] < S^*(n)[i]$, o algoritmo não estaria usando todas as moedas de forma gulosa, o que não é possível. Então $S(n)[i] > S^*(n)[i]$. Mas nesse caso, a solução ótima poderia ser melhorada, trocando-se as moedas de valor menor que fazem o troco para o valor $(S(n)[i] - S^*(n)[i]) \times c_i$ por menos moedas maiores, o que também é impossível. □

4 Resultados Importantes

Nessa seção apresentaremos alguns resultados da literatura que foram estudados durante este projeto.

4.1 Teoremas Estudados

Nesta seção vamos abordar teoremas que propõem condições necessárias e suficientes para que os conjuntos sejam considerados canônicos para sistemas de até 6 tipos de moedas.

Sabendo que sistemas de moedas com 1 ou 2 moedas são obviamente canônicos, Kozen e Zacks [4] trouxeram a caracterização de conjuntos não-canônicos com 3 tipos de moedas.

Teorema 4.1 (Kozen e Zacks [4]). *Um conjunto de moedas $M = (c_1, c_2, 1)$ é não-canônico se e somente se $0 > r > c_2 - q$ onde $c_1 = qc_2 + r$ para $0 \leq r < c_2$.*

A caracterização de conjuntos canônicos com 4 e 5 tipos de moedas foram propostas por Adamaszek e Cai [1].

Teorema 4.2 (Adamaszek e Cai [1]). *Um conjunto de moedas $M = (c_1, c_2, c_3, 1)$ é canônico se e somente se o sub-conjunto $(c_2, c_3, 1)$ é canônico e $S(nc_2) \leq n$ para $n = \lceil c_1/c_2 \rceil$.*

Teorema 4.3 (Adamaszek e Cai [1]). *Um conjunto de moedas* $M = (c_1, c_2, c_3, c_4, 1)$ *é canônico se e somente se um dos dois a seguir for verdadeiro:*

1. *O sub-conjunto* $(c_2, c_3, c_4, 1)$ *é canônico e* $S(nc_2) \leq n$ *para* $n = \lceil c_1/c_2 \rceil$*;*
2. $M = (2c_3, c_3 + 1, c_3, 2, 1)$ *e* $c_3 > 3$.

A caracterização de conjunto canônicos com 6 tipos de moedas foi proposta por Suzuki e Miyashiro [6].

Teorema 4.4 (Suzuki e Miyashiro [6]). *Um conjunto de moedas* $M = (c_1, c_2, c_3, c_4, c_5, 1)$ *é canônico se e somente se um dos dois a seguir for verdadeiro:*

1. *O sub-conjunto* $(c_2, c_3, c_4, c_5, 1)$ *é canônico e* $S(nc_2) \leq n$ *para* $n = \lceil c_1/c_2 \rceil$*;*
2. *O sub-conjunto* $(c_2, c_3, c_4, c_5, 1)$ *é não-canônico e* M *satisfaz um dos três a seguir, para* $\ell = \lceil c_2/c_4 \rceil$ *. Adicionalmente,* $S(lc_4) = lc_4 - c_2 + 1 - \lfloor (lc_4 - c_2)/c_5 \rfloor (c_5 - 1)$.
   (a) $M = (2c_3, c_3 + 1, c_3, 3, 2, 1)$ *e* $c_3 > 4$*;*
   (b) $M = (2c_3 - 1, c_5 + c_3 - 1, c_3, 2c_5 - 1, c_5, 1)$*,* $c_3 \geq 3c_5 - 1$*, e* $S(lc_4) \leq l$*;*
   (c) $M = (2c_3, c_5 + c_3, c_3, 2c_5, c_5, 1)$*,* $c_3 \geq 3c_5 - 1$*,* $c_3 \neq 3c_5$*,e* $S(S(lc_4) \leq l)$.

Pearson [5] provou o teorema a seguir, que caracteriza o menor contraexemplo possível de um sistema não-canônico.

Teorema 4.5 (Pearson[5]). *Seja* $M = (c_1, c_2, \ldots, c_{k-1}, 1)$ *um conjunto não-canônico e seja* n *o menor contraexemplo de* M*. Se* $S^*(n) = (\underbrace{0, \ldots, 0}_{0}, s^*_\ell, \ldots, s^*_r, \underbrace{0, \ldots, 0}_{0})$*, com* $s^*_\ell, s^*_r > 0$*, então* $S(c_{\ell-1} + 1) = (\underbrace{0, \ldots, 0}_{0}, s^*_\ell, \ldots, s^*_{r-1}, s^*_r + 1, s_{r+1}, \ldots, s_k)$.

4.2 Algoritmo de Verificação Canônico

O Teorema 4.5 apresentado anteriormente tem um papel fundamental na implementação de um algoritmo de tempo $O(k^3)$ apresentado por Pearson [5] que é capaz de decidir se um conjunto de moedas é canônico ou não. Ele é apresentado no Algoritmo 4.

Há também um algoritmo 5 de verificação de tempo $O(k^2)$ proposto por Xuan [2], que decide se um conjunto de moedas $M = (c_1, c_2, \ldots, c_{k-1}, 1)$ "justo" é canônico ou não, com $k \geq 6$. Primeiramente, vamos definir o que é um conjunto de moedas "justo".

Algoritmo 4 TestaCanonico(M, k)

Para $i = 2$ até k **faça**
  Para $j = i$ até k **faça**
    $S(c_{j-1} - 1) =$ TrocoGuloso$(c_{j-1} - 1, M, k)$
    seja $S(c_{j-1} - 1) = (s_1, \ldots, s_i, \ldots, s_j, \ldots, s_k)$
    construa $S^* = (0, \ldots, 0, s_i, \ldots, s_{j-1}, 0, \ldots, 0)$
    seja $n = (\sum_{m=j}^{k} c_m \times s_m) - 1$
    $S(n) =$ TrocoGuloso(n, M, k)
    Se $q(S^*) < q(S(n))$ **então**
      Devolve falso
Devolve verdadeiro

Definição 4.1. *Um conjunto de moedas $M = (c_1, c_2, \ldots, c_{k-1}, 1)$ é considerado "justo" se ele não tem contraexemplo menor que c_1.*

Exemplo: sejam os seguintes dois conjuntos de moedas não-canônicos $M_1 = (11, 10, 7, 1)$ e $M_2 = (50, 10, 7, 1)$. Note que 14 é o contraexemplo deles, ou seja, $S_1(14) = S_2(14) = (1, 0, 0, 3)$ e $S_1^*(14) = S_2^*(14) = (0, 0, 2, 0)$. Assim, é possível verificar que M_1 é justo, mas M_2 não, pois o contraexemplo $14 > 11$.

Usando os teoremas apresentados anteriormente, Xuan [2] propôs o Algoritmo 5.

Algoritmo 5 TestaCanonicoJusto(M, k)

Se $0 > r > M[k-1] - p$ com $M[k-2] = pM[k-1] + r$ **então**
  Devolve falso
Senão
  Para $i = 1$ até k **faça**
    Para $j = 1$ até k **faça**
      Se $M[i] + M[j] > M[k+1]$ **e** $q(S(M[i] + M[j])) > 2$ **então**
        Devolve falso
  Devolve verdadeiro

Esse algoritmo pode lidar com a maioria de conjuntos de moedas justos, exceto um pequeno número de conjuntos não-canônicos, que na maioria das vezes são progressões aritméticas.

5 Programação Dinâmica

A programação dinâmica é uma técnica utilizada para lidar com problemas de otimização, procurando pela melhor solução dentre várias possíveis ao resolver problemas combinando soluções para subproblemas. Ela se aplica quando os subproblemas se sobrepõem, ou seja, quando os subproblemas compartilham subsubproblemas.

Um algoritmo de programação dinâmica resolve cada subsubproblema apenas uma vez e depois armazena sua resposta em uma tabela, evitando assim o trabalho de recalcular a resposta toda vez que resolve cada subsubproblema.

Ela segue quatro etapas: caracterizar a estrutura da solução ideal, definir recursivamente o valor da solução, calcular esse valor de forma eficiente e construir a solução a partir das informações calculadas.

Uma solução encontrada por Wright [7] para o Problema do Troco envolve o uso de um algoritmo de Programação Dinâmica que não só é capaz de solucionar o problema, como também o resolve em tempo $O(kn)$, que é pseudo-polinomial, pois está em função do valor numérico n e não da quantidade de bits que o representa.

Neste texto, optamos por não aprofundar a programação dinâmica, pois nosso foco foi a compreensão do algoritmo guloso no contexto do Problema do Troco e dos conjuntos canônicos. A razão para essa escolha reside no fato de que a programação dinâmica, com suas complexidades e estratégias mais avançadas, requer um nível mais profundo de conhecimento técnico.

Entretanto, é importante reconhecer a existência e a relevância da programação dinâmica, pois ela desempenha um papel fundamental na resolução de uma ampla gama de problemas computacionais. Para aqueles que desejam explorar mais a fundo esse campo, recomendamos o livro *Introduction to Algorithms*, de Cormen *et al.* [3] e o artigo *The change-making problem*, de Wright [7].

6 Observações Finais

Neste projeto de iniciação científica, exploramos conceitos fundamentais de análise de algoritmos, com foco no Problema do Troco. Começamos definindo algoritmos e discutindo sua importância, enfatizando a prova de corretude e análise do tempo de execução, bem como a busca pela solução ótima, especialmente aplicadas ao Problema do Troco.

Destacamos o algoritmo guloso como uma ferramenta eficaz para resolver o problema, introduzimos os conjuntos canônicos de moedas, que desempenham um papel fundamental na análise da solução ótima.

Além disso, mencionamos a programação dinâmica como uma abordagem alternativa

para resolver o Problema do Troco de forma ótima em tempo Pseudo-polinomial.

Esta iniciação científica proporcionou uma base sólida para compreender os algoritmos gulosos e os princípios essenciais da análise de algoritmos. Dominar esses conceitos é fundamental para o desenvolvimento de soluções eficazes em diversas áreas da computação.

Referências

[1] A. Adamaszek and M. Adamaszek. Combinatorics of the change-making problem. *European Journal of Combinatorics*, 31(1):47–63, 2010. doi:10.1016/j.ejc.2009.05.002.

[2] X. Cai. Canonical coin systems for change-making problems. In *2009 Ninth International Conference on Hybrid Intelligent Systems*, volume 1, pages 499–504, 2009. doi:10.1109/HIS.2009.103.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. ISBN 0262033844.

[4] D. Kozen and S. Zaks. Optimal bounds for the change-making problem. *Theoretical Computer Science*, 123(2):377–388, 1994. ISSN 0304-3975. doi:g/10.1016/0304-3975(94)90134-1.

[5] D. Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33(3):231–234, 2005. ISSN 0167-6377. doi:10.1016/j.orl.2004.06.001.

[6] Y. Suzuki and R. Miyashiro. Characterization of canonical systems with six types of coins for the change-making problem. *Theoretical Computer Science*, 955:113822, 2023. ISSN 0304-3975. doi:10.1016/j.tcs.2023.113822.

[7] J. W. Wright. The change-making problem. *Journal of the ACM*, 22(1):125—-128, 1975. ISSN 0004-5411. doi:10.1145/321864.321874.