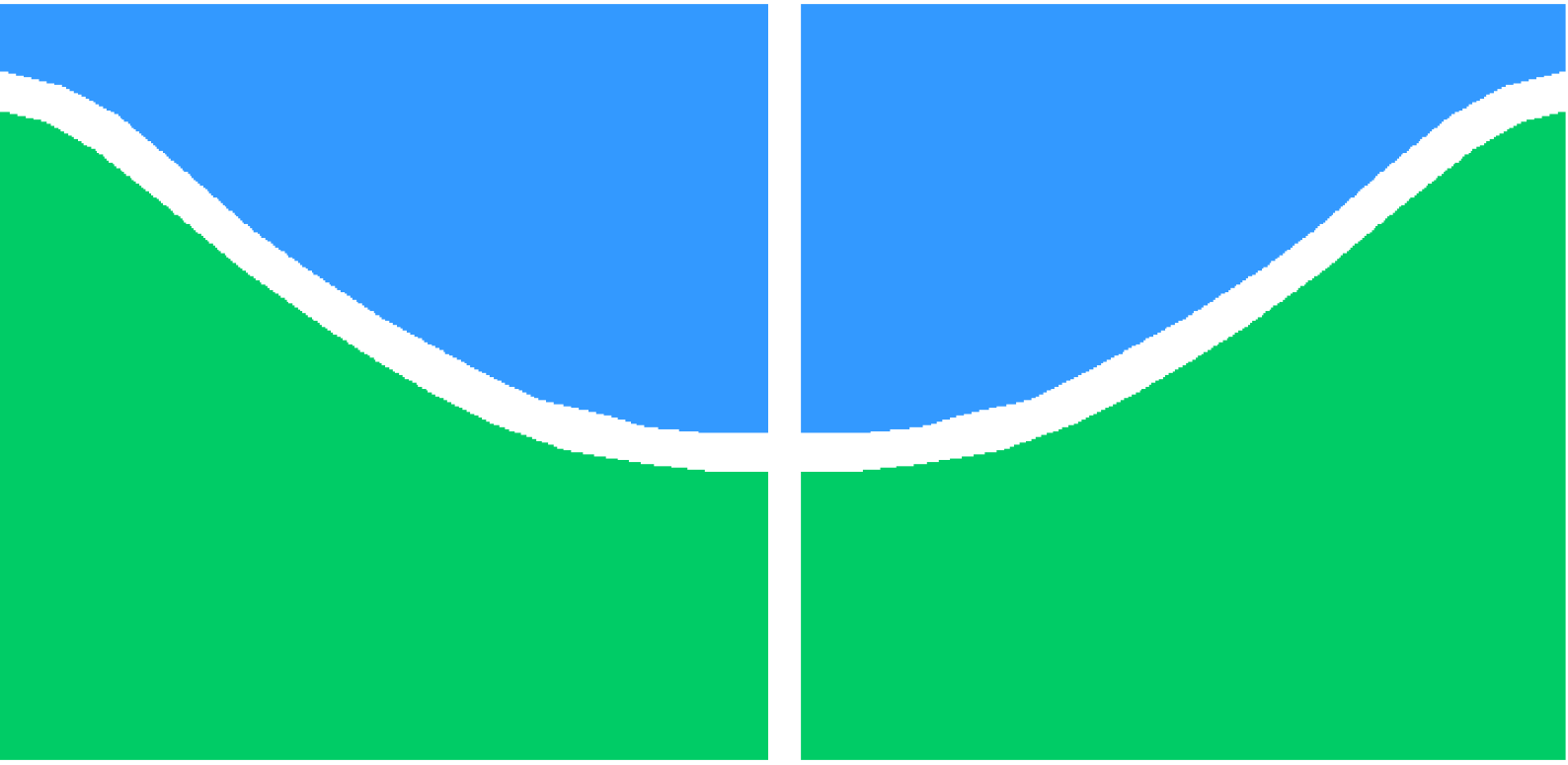

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Engenharia de Software

Comparativo entre a estratégia gulosa e a programação dinâmica para o problema do troco com sistemas de moedas canônicos

Autor: Lucas Vasconcelos Mattioli

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2018

Lucas Vasconcelos Mattioli

Comparativo entre a estratégia gulosa e a programação dinâmica para o problema do troco com sistemas de moedas canônicos

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2018

Lucas Vasconcelos Mattioli

Comparativo entre a estratégia gulosa e a programação dinâmica para o problema do troco com sistemas de moedas canônicos/ Lucas Vasconcelos Mattioli. – Brasília, DF, 2018-

57 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2018.

1. problema do troco. 2. sistemas canônicos. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Comparativo entre a estratégia gulosa e a programação dinâmica para o problema do troco com sistemas de moedas canônicos

CDU 02:141:005.6

Lucas Vasconcelos Mattioli

Comparativo entre a estratégia gulosa e a programação dinâmica para o problema do troco com sistemas de moedas canônicos

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 11 de dezembro de 2018 – Data da aprovação do trabalho:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. MsC. Daniel Saad Nogueira Nunes
Convidado 1

Prof. Felipe Duerno do Couto Almeida
Convidado 2

Brasília, DF
2018

Agradecimentos

Agradeço à minha família pelo apoio durante toda minha graduação. Agradeço a todos os professores que fizeram parte da minha graduação de alguma forma, em especial ao professor Edson Alves, que me apoiou, incentivou e inspirou em diversas situações. Agradeço, também, aos meus colegas de curso pelos momentos, experiências e conhecimentos compartilhados. Por fim, um agradecimento especial aos meus amigos do grupo SACC, os quais mudaram minha visão de mundo.

Resumo

O problema do troco é uma especialização do problema da mochila, sendo possível resolvê-lo com um algoritmo de programação dinâmica. É possível, também, resolver o problema utilizando um algoritmo guloso quando o sistema de moedas do problema é canônico. Este trabalho analisa e compara os tempos de execução dos algoritmos de programação dinâmica, guloso e de um algoritmo de determinação da canonicidade em cima de sistemas de moedas gerados pseudo-aleatoriamente. Os algoritmos foram implementados e seus tempos mensurados na linguagem de programação C++, seguindo o padrão C++14. Foi verificado que, apenas para sistemas de moedas com tamanhos pequenos (até 76 moedas), o algoritmo de programação dinâmica se mostra mais eficiente que os algoritmos guloso e de determinação de canonicidade, este necessário para garantir a canonicidade do sistema e assegurar a correção do algoritmo guloso.

Palavras-chaves: problema do troco. sistemas canônicos. programação dinâmica.

Abstract

The coin change problem is a specialization of the knapsack problem, being possible to solve it with a dynamic programming algorithm. It is also possible to solve the problem using a greedy algorithm when the coin system is canonical. This work analyzes and compares the execution times of dynamic programming, greedy and canonicity determination algorithms on top of pseudo-randomly generated coin systems. The algorithms were implemented in the C++ programming language, following the C++ 14 standard. It was verified that, solely for coin systems with small sizes (up to 76 coins), the dynamic programming algorithm is more efficient than the greedy one, along with the canonicity determination algorithm, which is used to guarantee the canonicity of the system and enable the use of the greedy algorithm.

Key-words: coin change. canonical coin systems. dynamic programming.

Lista de ilustrações

Sumário

Introdução

O problema do troco é inspirado na aversão a carregar moedas que o ser humano cultua. Ao fazer uma compra, caso o dinheiro entregue pelo comprador seja maior do que o preço dos produtos a serem comprados, o funcionário do estabelecimento deve retornar uma quantia, esta denominada troco. Muitas vezes, as plataformas eletrônicas que auxiliam os funcionários apenas expelam o valor do troco a ser entregue, não especificando um conjunto de notas e/ou moedas, o que resulta em compradores saindo do estabelecimento com o bolso cheio de moedas indesejadas.

Assim, pode ser interessante conseguir determinar um conjunto de moedas em que o número de moedas presentes é o menor possível. A tentativa natural de resolução do problema é buscar a maior moeda possível que não ultrapassa o troco atual e repetir até que o troco esteja completo. Acontece que esse processo, base do algoritmo guloso para o problema do troco, não determina soluções corretas em todos os casos, mesmo determinando soluções corretas para muitos casos. Em contrapartida, algoritmos de programação dinâmica resolvem o problema com garantia de correção, mas com complexidades assintóticas de tempo de execução e memória piores. Dessa forma, entender os motivos por trás da correção de algoritmos gulosos em alguns casos nos ajuda a traçar estratégias diferentes na resolução do problema, já que o tempo de execução do mesmo e a correção dos algoritmos de programação dinâmica podem ser combinados para atingir uma melhor solução para um dado contexto.

Existem sistemas de moedas (como o real, por exemplo) em que a aplicação do algoritmo guloso funciona corretamente em todo e qualquer caso. Esses sistemas são denominados canônicos, e a canonicidade de um sistema canônico com m moedas pode ser determinada com o auxílio de um algoritmo $O(m^2)$ como proposto por Cai (2009), o que possibilita a busca por boas estratégias para a resolução do problema do troco.

Perguntas que surgem dada a existência do algoritmo de determinação de canonicidade são: para um sistema de moedas qualquer, é mais eficiente em termos de tempo de execução sempre utilizar o algoritmo de programação dinâmica, mesmo este tendo complexidades assintóticas de memória e tempo de execução mais elevados em relação ao algoritmo guloso? Ou será que é melhor determinar a canonicidade do sistema de moedas e: caso ela seja confirmada, utilizar o algoritmo guloso para a resolução do problema; caso não seja, utilizar o algoritmo de programação dinâmica? Essas perguntas são motivação para a realização de experimentações quanto aos tempos de execução considerando o algoritmo guloso, o algoritmo de programação dinâmica e o algoritmo de determinação de canonicidade e as diversas possíveis estratégias para a resolução do problema do troco.

Objetivos do Trabalho

Este trabalho tem como objetivo geral entender propriedades relacionadas ao problema do troco, como:

- Comparar tempos de execução dos algoritmos de programação dinâmica e guloso, este aliado ao algoritmo de canonicidade, para a resolução do problema do troco, levando em conta sistemas de moedas canônicos e não necessariamente canônicos sorteados pseudo-aleatoriamente.
- Determinar a taxa de acertos do algoritmo guloso em relação a sistemas de moedas não canônicos: isto é, sistemas que não garantem que o algoritmo guloso sempre retorna a resposta correta para o problema do troco.

Como objetivos específicos, tem-se:

- Implementar o algoritmo de determinação de canonicidade de sistemas de moedas utilizando a linguagem de programação C++.
- Implementar os algoritmos de programação dinâmica e guloso para resolução do problema do troco utilizando a linguagem de programação C++.
- Realizar experimentos de mensuração do tempo de execução dos algoritmos de programação dinâmica e guloso, este aliado ao algoritmo de canonicidade, para a resolução do problema do troco, levando em conta sistemas de moedas canônicos e não necessariamente canônicos sorteados pseudo-aleatoriamente.

Estrutura do Trabalho

Este trabalho está dividido em 4 capítulos. O Capítulo 1 apresenta a fundamentação teórica necessária para entendimento do capítulos posteriores, incluindo a descrição do problema do troco e seus algoritmos relacionados. O Capítulo 2 descreve a metodologia seguida para a realização do trabalho. O Capítulo 3 apresenta os resultados e as discussões realizadas em cima dos experimentos propostos no Capítulo 2. O Capítulo 4 faz as considerações finais do trabalho e propõe trabalhos futuros a respeito do tema.

1 Fundamentação Teórica

Este capítulo tem como objetivo dar ao leitor deste trabalho toda a fundamentação teórica necessária para entendimento dos experimentos realizados, seus métodos, resultados e conclusões. A Seção 1.1 apresenta a definição de algoritmos. A Seção 1.2 apresenta os principais paradigmas de programação, incluindo o guloso e a programação dinâmica. A Seção 1.3 mostra o problema da mochila, suas variações, incluindo o problema do troco, e suas soluções, as quais fazem uso da programação dinâmica.

1.1 Algoritmos

Um algoritmo, segundo Dijkstra (1971), corresponde a uma descrição de um padrão comportamental, evidenciado em termos de um conjunto finito de ações. Quando a operação $a + b$ é executada, percebe-se um mesmo padrão comportamental, mesmo que a mesma seja realizada para diferente valores de a e b.

Informalmente, Knuth (1968) define cinco propriedades como requerimentos para a composição de um algoritmo computacional:

- **Finitude**: um algoritmo deve sempre terminar depois de um número finito de passos;
- **Defitinividade**: cada passo de um algoritmo deve ser precisamente definido;
- **Entradas**: objetos vindos de conjuntos bem definidos que são dados antes do ínicio do algoritmo;
- **Saídas**: objetos que são expelidos pelo algoritmo em seu término;
- **Eficácia**: todas as operações a serem executadas pelo algoritmo devem ser suficientemente básicas de tal forma que elas possam ser executadas por um humano de forma exata e em uma quantidade finita de tempo com um papel e uma caneta.

Muitas vezes, um algoritmo computacional que sempre computa a melhor resposta possível para um dado problema é lento em relação à necessidade do problema. O objetivo de uma heurística em um algoritmo é produzir uma solução não necessariamente ótima para um dado problema em uma quantidade de tempo razoável. A solução, mesmo não sendo a melhor possível dado o espaço de busca do problema, pode ser valiosa para o contexto, dado o possível ganho na relação entre correção e tempo de execução em relação à solução demorada mas ótima (PEARL, 1984).

No contexto deste trabalho, o Algoritmo Guloso para resolução do problema do troco, o qual será melhor detalhado em seções posteriores, pode ser considerado como uma heurística em certas situações. Como será visto, seu tempo de execução é, assintoticamente, superior ao do Algoritmo de Programação Dinâmica. Entretanto, sua correção é garantida apenas para sistemas de moedas canônicos, servindo como heurística nos casos dos sistemas não serem canônicos.

1.2 Paradigmas de Solução de Problemas

Ao tentar resolver um problema computacional, várias estratégias randômicas podem vir a cabeça do solucionador. Muitas vezes, o problema a ser solucionado possui uma solução *ad hoc*: isto é, uma solução que é feita para aquele problema específico, sendo difícil generalizá-la. Entretanto, muitos problemas compartilham soluções que seguem um certo tipo de padrão e essas soluções podem ser encaixadas em grupos de solução de problemas, os chamados paradigmas de solução de problemas. Esta seção tem como objetivo exemplificar os principais paradigmas de solução de problemas: busca completa, algoritmos gulosos, divisão e conquista e programação dinâmica.

1.2.1 Busca Completa

Uma solução que usa o paradigma de busca completa explora todo o espaço de busca que está atrelado ao problema. Em geral, as soluções de busca completa são aplicadas em problemas de otimização, onde todas as possibilidades de estado são testadas e a que possuir o melhor valor baseado na otimização requerida é escolhida como ótima.

Usemos um problema para exemplificar o paradigma: dado um inteiro N, deseja-se encontra uma permutação $P = \{P_1, P_2, ..., P_N\}$ de $\{1, 2, ...N\}$ tal que seu coeficiente de diferença seja o maior possível. O coeficiente de diferença de uma permutação $K = \{K_1, K_2, ..., K_N\}$ de $\{1, 2, ..., N\}$ é definido na Equação 1.1.

$$\sum_{i=1}^{N-1} |K_{i+1} - K_i| \tag{1.1}$$

Intuitivamente, a solução mais simples que se pode pensar é testar toda permutação p e escolher a que possui o maior coeficiente de diferença: e é exatamente isso que uma solução sob o paradigma de busca completa faria. É fácil ver que essa solução é correta, afinal, todas as possibilidades são testadas. As principais vantagens dessa solução (e de todas de busca completa) são a garantia de correção e, em geral, a facilidade de codificação da solução. O maior ponto negativo, em geral, é a complexidade assintótica de tempo de execução e de memória: no problema exemplo, é notável que existem $N!$ possíveis permutações de $\{1, 2, ..., N\}$; como cada uma delas é testada uma vez, a complexidade

assintótica de tempo de execução da solução é pelo menos $\Omega(N!)$, a qual para $N > 20$ levaria alguns anos para ser computada.

Ainda no problema, o que resta é gerar todas as permutações possíveis do conjunto $\{1, 2, ..., N\}$. Para isso, uma técnica bastante conhecida denominada *backtracking* pode ser utilizada. O *backtracking* constrói os candidatos a serem soluções (no caso do problema exemplificado: as permutações) incrementalmente e abandona os candidatos assim que eles forem dados como inválidos (no caso do problema, poderíamos desconsiderar as permutações parciais que não alcançariam o melhor valor já encontrado até então), com auxílio da recursão (KNUTH, 1968).

Uma forma de visualizar o processo do *backtracking* é imaginar sua árvore enraizada de recursão, onde a raiz representa uma permutação parcial vazia, os nós intermediários representam as permutações parciais em construção e as folhas representam todas as possíveis $N!$ permutações. Para solucionar o problema, assim que o *backtracking* estiver em uma folha que representa uma permutação $p = \{p_1, p_2, ..., p_n\}$, basta iterar por todas as $N - 1$ posições de 1 a $N - 1$ e somar os valores $|p_{i+1} - p_i|$ e atualizar o melhor valor já encontrado até então caso o valor atual seja maior que o mesmo. Como esse último passo de processar uma permutação completa do algoritmo tem complexidade assintótica $O(N)$, o algoritmo final para solucionar o problema do maior coeficiente de diferença tem complexidade de tempo de execução $O(N \cdot N!)$ e, com uma implementação cuidadosa, $O(N)$ de memória.

1.2.2 Algoritmos Gulosos

Segundo Cormen (2011),

> "Algoritmos para problemas de otimização normalmente passam por uma sequência de etapas e cada etapa tem um conjunto de escolhas. Para muitos problemas de otimização, é um exagero utilizar programação dinâmica para determinar as melhores escolhas: algoritmos mais simples e mais eficientes servirão. Um algoritmo guloso sempre faz a escolha que parece ser a melhor no momento em questão. Isto é, faz uma escolha localmente ótima, na esperança de que essa escolha leve a uma solução globamente ótima".

Levando em conta a afirmação acima, os algoritmos gulosos tendem a ser mais eficientes por não analisar todos os estados posteriores possíveis para cada escolha da etapa atual. Fazendo a escolha que parece a melhor no momento, uma grande parte dos caminhos de soluções é ignorada, resultando em uma travessia em um conjunto relativamente pequeno de soluções (CORMEN, 2011).

Apesar da eficiência, os algoritmos gulosos nem sempre retornam a resposta correta para um dado problema. Em outras palvras, as respostas encontradas pelos algoritmos

gulosos não são necessariamente ótimas globalmente, devido às escolhas ótimas locais. Este fato, também, está atrelado à natureza de alguns tipos de problemas: alguns deles não possuem soluções gulosas de forma alguma. Para os problemas que possuem soluções gulosas globalmente ótimas, achar o algoritmo guloso que o resolve muitas vezes não é uma tarefa simples: em geral, dado um estado no espaço de busca do problema, muitas observações devem ser feitas para ser possível decidir quais escolhas locais tomar para que a solução global ótima possa ser alcançada.

1.2.3 Divisão e Conquista

Soluções de divisão e conquista funcionam de modo a quebrar um problema grande em subproblemas menores com mesma estrutura, resolver esses subproblemas recursivamente e combinar suas soluções. Os problemas e seus subproblemas são quebrados em partes menores até que seja possível resolver um subproblema de forma muito simples, sem que o mesmo precise ser quebrado em partes menores ou mais fáceis. Em geral, a semântica dos subproblemas é igual à do problema grande (isto é, suas entradas e saídas seguem o mesmo mapeamento do problema grande), restando ao solucionador decifrar como combinar os vários retornos dos subproblemas para que a solução do problema grande possa ser obtida.

Um clássico exemplo de algoritmo que utiliza o paradigma de divisão e conquita é o *merge sort*. O problema resolvido pelo *merge sort* é a ordenação de elementos: dado um vetor $A = \{A_1, A_2, ..., A_N\}$ de N elementos, deseja-se ordená-los, sem perda de generalidade, de forma não-decrescente; isto é, deseja-se obter uma permutação $P = \{P_1, P_2, ..., P_N\}$ de $\{1, 2, ..., N\}$ de tal forma que $A_{P_i} \leq A_{P_{i+1}}$ para todo para $1 \leq i \leq N - 1$.

Levando em conta o problema de ordenação, o *merge sort* (e os algoritmos de divisão e conquista) é dividido em duas partes:

1. **Divisão**: dividir o vetor A em dois vetores com diferença de tamanho no máximo 1 e chamar o *merge sort* para cada um deles, recursivamente;

2. **Conquista**: obter o vetor ordenado combinando os dois vetores (já ordenados, por definição) retornados pelas recursões chamadas no passo de divisão.

Para enxergar melhor o funcionamento do *merge sort* podemos analisar o Algoritmo 1.

Algoritmo 1: Merge Sort

```
1  function: MergeSort(A)
2  input: an array A = A_1, A_2, ..., A_N of size N
3  output: the array A sorted
4
5  if size(A) = 1:
6   return A
7
8  B ← A[0..⌊N/2⌋ − 1]
9  C ← A[⌊N/2⌋..N − 1]
10
11 B ← MergeSort(B)
12 C ← MergeSort(C)
13
14 A ← {}
15 B_ptr ← 0
16 C_ptr ← 0
17 while B_ptr < size(B) or C_ptr < size(C), do:
18   if C_ptr == size(C) or B[B_ptr] < C[C_ptr]:
19     insert B[B_ptr] at A's ending
20     B_ptr ← B_ptr + 1
21   else:
22     insert C[C_ptr] at A's ending
23     C_ptr ← C_ptr + 1
24
25 return A
```

O caso base da recursão do algoritmo acontece nas linhas 4 e 5: se temos um vetor de tamanho 1, então obviamente o próprio vetor já está ordenado, não sendo necessário reduzir ainda mais para tomar essa decisão. Da linha 7 à linha 11 temos o passo de **Divisão**: os dois vetores (B e C, nesse caso) representando, cada um, uma das metades de A são ordenados recursivamente. Por fim, a partir da linha 13, o passo de **Conquista** acontece: ordenamos o vetor A com base nos vetores B e C.

Para analisar a complexidade final do algoritmo é necessário entender quantos níveis de recursão são alcançados durante o *merge sort* e quanto de trabalho é feito para uma entrada de tamanho N em uma chamada do *merge sort*.

Em relação aos níveis de recursão, é possível perceber que para um vetor A inicial de tamanho N, em cada uma das próximas chamadas recursivas, os subvetores que o dividem na metade terão tamanho aproximadamente $\frac{N}{2}$. Nas chamadas recursivas dos subvetores, os subsubvetores terão tamanho aproximadamente $\frac{N}{4}$ e assim por diante. É visível que para um dado subvetor que foi dividido k vezes a partir de um vetor inicial de tamanho N, seu tamanho será aproximadamente $\frac{N}{2^k}$. Como a recursão acaba no caso base, em que o tamanho do vetor atual é 1, desejamos saber o limite superior para k de tal forma que $\frac{N}{2^k} = 1$. Com simples manipulação algébrica, é possível obter que o limite

superior de k é aproximadamente $\log_2 N$.

Para a quantidade trabalho feita para uma entrada de tamanho N, precisamos analisar o passo de **Conquista**. Como no *loop while* exatamente uma das variáveis B_{ptr} e C_{ptr} tem seu valor incrementado em 1 e os tamanhos de B e C somados resultam em N, então o *loop while* será executado no máximo $O(N)$ vezes. Supondo que a inserção de um elemento qualquer no final de A tenha complexidade de tempo $O(1)$, a quantidade de trabalho realizada em uma chamada do *merge sort* é $T(N)$.

Cada nível de recursão k mantém aproximadamente 2^k instâncias de chamada do *merge sort*, e considerando que cada uma delas possui um vetor de tamanho aproximadamente $\frac{N}{2^k}$, então tem-se a quantidade de trabalho total F por nível dada pela Equação 1.2.

$$F \approx 2^k \frac{N}{2^k} \approx N \tag{1.2}$$

Assim, considerando que cada nível realiza $T(N)$ de trabalho e existem no máximo aproximadamente $\log_2 N$ níveis de recursão, a complexidade do algoritmo final de tempo de execução e memória é $O(N \log_2 N)$.

1.2.4 Programação Dinâmica

Segundo Cormen (2011)

> "A programação dinâmica resolve problemas combinando as soluções para subproblemas. A programação dinâmica se aplica quando os subproblemas se sobrepõem, isto é, quando os subproblemas compartilham subsubproblemas. Um algoritmo de programação dinâmica resolve cada subsubproblema apenas uma vez e depois grava sua resposta em algum lugar, evitando assim, o trabalho de recalcular toda vez que resolver cada subsubproblema. Em geral, aplicamos a programação dinâmica a problemas de otimização. Tais problemas podem ter muitas soluções possíveis. Cada solução tem um valor, e desejamos encontrar uma solução com o valor ótimo."

Basicamente, a resolução de um problema utilizando programação dinâmica consiste em 4 etapas (CORMEN, 2011)

1. Caracterizar a estrutura de uma solução ótima.
2. Definir recursivamente o valor de um solução ótima.
3. Calcular o valor de uma solução, utilizando uma estratégia *top down* ou uma estratégia *bottom up*.
4. Construir uma solução ótima com as informações calculadas.

Quando definidas corretamente, as soluções de programação dinâmica expelem respostas ótimas para os problemas a serem resolvidos. Essa propriedade se assemelha com as soluções do paradigma de busca completa, apresentado na Seção 1.2.1. O paradigma de programação dinâmica consegue reduzir o espaço de busca agrupando estados que compartilham propriedades semelhantes e combinando-os em um novo estado. Estes agrupamentos resultam em um espaço de busca menor mas correto semanticamente. As soluções de programação dinâmica, então, percorrem todos os estados possíveis, assim como no paradigma de busca completa, com a única diferença de que o espaço de busca é menor do que o original.

1.3 Problema da Mochila

1.3.1 Problema da Mochila Binária

O problema da mochila binária é um problema de otimização combinatória que pode ser visto da seguinte forma: tem-se N objetos e uma mochila com capacidade de suportar até C unidades de massa. Cada objeto possui um valor V_i e uma massa M_i e deseja-se colocar um subconjunto dos objetos na mochila de tal forma que o valor total dos objetos colocados seja maximizado e a massa total dos objetos colocados seja menor ou igual a C (ANDONOV; POIRRIEZ; RAJOPADHYE, 2000).

Em outras palavras, são dados: um inteiro não-negativo C, um inteiro não-negativo N e dois conjuntos $V = \{V_1, V_2, ..., V_N\}$ e $M = \{M_1, M_2, ..., M_N\}$; e deseja-se escolher um subconjunto E de $\{1, 2, ..., N\}$ de tal forma a maximizar a Equação 1.3

$$\sum_{i \in E} V_i \tag{1.3}$$

sujeita à Equação 1.4.

$$\sum_{i \in E} M_i \leq C \tag{1.4}$$

Uma possível forma de resolução do problema é utilizando o paradigma de busca completa com o emprego da técnica *backtracking*: nesse caso, todos os subconjuntos de E são escolhidos como candidato e, para cada um, verifica-se se o valor total é o maior já encontrando até agora caso a massa total atual não ultrapasse C. O ponto negativo de tal estratégia é sua complexidade assintótica de tempo de execução: já que qualquer conjunto de tamanho K possui 2^K subconjuntos, então é fácil ver que a complexidade do algoritmo de tempo de execução é pelo menos $\Omega(2^N)$ enquanto pode-se atingir, com uma implementação cuidadosa, uma complexidade de memória $O(N)$.

Outra alternativa para solucionar o problema é utilizando programação dinâmica com complexidade de tempo de execução e memória $O(NC)$ (HOFFMAN; RALPHS,

2012). A função de programação dinâmica $F(i, T)$ pode ser vista como: qual o maior valor total possível de ser obtido se apenas os i primeiros elementos de $\{1, 2, ..., N\}$ forem considerados, levando em conta que tem-se uma mochila de capacidade T? As transições da função podem, intituivamente, ser divididas em dois cenários: inserir ou não o i-ésimo objeto na mochila. Caso o objeto não seja inserido, então consideramos a solução para os $i - 1$ primeiros elementos com uma mochila ainda com capacidade T e, caso contrário, consideramos a solução para os $i-1$ primeiros elementos mas agora como uma mochila de capacidade $T - M_i$. Dessa forma, o resultado para a chamada $F(N, C)$ retorna a resposta correta para o problema. A definição da recorrência pode ser conferida na Equação 1.5:

$$F(i,T) = \begin{cases} 0, & \text{se } i = 0 \text{ ou } T < 0, \\ \max(F(i-1,T), F(i-1,T-M_i) + V_i), & \text{caso contrário} \end{cases} \tag{1.5}$$

O Algoritmo 2 referente à Equação 1.5 pode ser conferido a seguir:

Algoritmo 2: Mochila Binária - Programação Dinâmica

```
1  function: F(i, T)
2  input: a non-negative integer T, a non-negative integer i, a set
       V = {V_1, V_2, ..., V_N}
3  and a set M = {M_1, M_2, ..., M_N}
4  output: a non negative integer representing the answer for the knapsack
       problem
5
6  if i = 0 or T < 0, then:
7   return 0
8
9  if state (i,T) is already calculated, then:
10  return answer[i][T]
11
12 total ← max(F(i-1,T), F(i-1,T-M_i) + V_i)
13 answer[i][T] ← total
14
15 return total
```

1.3.2 Problema da Mochila Sem Limites

O problema da mochila sem limites é bastante parecido com o problema da mochila binária: as entradas para o problema são as mesmas, a única diferença é que, agora, supõe-se que existe uma quantidade ilimitada de cada um dos N objetos; isto é, pode-se colocar qualquer quantidade inteira de objetos do tipo i na mochila desde que a capacidade C não seja ultrapassada. O ajuste na recorrência da solução de programação dinâmica para considerar a nova condição é simples de ser feito, bastando apenas não desconsiderar o

elemento atual quando o mesmo for colocado na mochila:

$$G(i,T) = \begin{cases} 0, & \text{se } i = 0 \text{ ou } T < 0, \\ \max(G(i-1,T), G(i,T-M_i) + V_i), & \text{caso contrário} \end{cases} \tag{1.6}$$

O Algoritmo 3 referente à Equação 1.6 pode ser conferido a seguir:

Algoritmo 3: Mochila Sem Limites - Programação Dinâmica

```
1  function: G(i, T)
2  input: a non-negative integer T, a non-negative integer i, a set
       V = {V_1, V_2, ..., V_N}
3  and a set M = {M_1, M_2, ..., M_N}
4  output: a non negative integer representing the answer for the knapsack
       problem
5
6  if i = 0 or T < 0, then:
7   return 0
8
9  if state (i,T) is already calculated, then:
10  return answer[i][T]
11
12 total ← max(G(i-1,T), G(i,T-M_i) + V_i)
13 answer[i][T] ← total
14
15 return total
```

1.3.3 Especialização do Problema da Mochila Sem Limites: Problema do Troco

O problema do troco pode ser modelado como uma instância do problema da mochila sem limites com a adição de uma nova condição: agora, queremos obter o menor valor possível (no caso, número de moedas) e que toda a capacidade C da mochila seja utilizada. A nova recorrência de programação dinâmica pode ser vista na Equação 1.7:

$$H(i,T) = \begin{cases} -\infty, & \text{se } i = 0 \text{ e } T \neq 0, \\ 0, & \text{se } i = 0 \text{ ou } T < 0, \\ \max(H(i-1,T), H(i,T-M_i) + V_i), & \text{caso contrário} \end{cases} \tag{1.7}$$

Assim, dado um sistema de moedas $A = \{C_1, C_2, ..., C_m\}$ e um troco W, pode-se considerar: que tem-se uma mochila de capacidade W, que $M = A$ e que $V = \{-1, -1, ..., -1\}$. A resposta para o problema seria, então, $-H(m, W)$. É importante notar a negação do retorno na chamada da função, pois no problema do troco queremos minimizar a quantidade de moedas a serem utilizadas e no problema da mochila queremos

maximizar a quantidade de valor ganho; assim, levando em conta que os valores V são negativos, a função min passa a ser equivalente à função max.

Mais detalhadamente, pode-se modelar o problema do troco da seguinte forma: dado um sistema de moedas $A = \{C_1, C_2, ..., C_m\}$, com $C_1 = 1$, $C_1 < C_2 < ... < C_m$ e C_i representando o valor da moeda de tipo i em $R\$$, e um valor inteiro não negativo W em $R\$$ de troco a ser entregue, deseja-se escolher um conjunto de m valores inteiros não negativos $(E_1, E_2, ..., E_m)$ de tal forma que $\sum_{i=1}^{m} E_i \cdot C_i = W$ e $\sum_{i=1}^{m} E_i$ seja o menor possível (NIEWIAROWSKA; ADAMASZEK, 2010).

Para resolver o problema do troco computacionalmente, existem, basicamente, duas alternativas: utilizando um algoritmo guloso ou utilizando um algoritmo de programação dinâmica. O algoritmo guloso tradicional (JENKINS, 2004) tem complexidade assintótica $O(m)$ em tempo de execução, onde m é a quantidade de moedas de um dado conjunto A, e $O(1)$ em memória. O algoritmo de programação dinâmica tradicional (KLEINBERG; TARDOS, 2011) tem complexidade assintótica $O(W \cdot m)$ em tempo de execução e $O(W)$ em memória, onde m é a quantidade de moedas de um dado conjunto A e W é o valor de troco a ser entregue.

Um conjunto de moedas escolhidas $(E_1, E_2, ..., E_m)$ pelo algoritmo guloso em cima do sistema A para um dado troco W é denominado como uma representação gulosa de W em A, que será denotada como $GUL_A(W)$, e tem seu tamanho detonado por $|GUL_A(W)| = \sum_{i=1}^{m} E_i$.

Um conjunto de moedas escolhidas $(E_1, E_2, ..., E_m)$ pelo algoritmo de programação dinâmica em cima do sistema A para um dado troco W é denominado como uma representação ótima de W em A, que será denotada como $OPT_A(W)$, e tem seu tamanho detonado por $|OPT_A(W)| = \sum_{i=1}^{m} E_i$.

Um sistema de moedas é denominado canônico caso, para qualquer troco dado, a quantidade de moedas dada como resposta pelo algoritmo guloso seja igual à quantidade de moedas dada como resposta pelo algoritmo de programação dinâmica. Se algum troco causar uma resposta maior por parte do algoritmo guloso em relação ao de programação dinâmica, então ele é denominado um contraexemplo do sistema de moedas. É possível determinar se um dado sistemas de moedas é canônico em $O(m^2)$ (onde m é a quantidade de moedas do sistema) utilizando um algoritmo de canonicidade proposto por Cai (2009).

Algoritmo Guloso para o Problema do Troco

Como dito antes, para um sistema de moedas $A = \{C_1, C_2, ..., C_m\}$, com $C_1 = 1$ e $C_1 < C_2 < ... < C_m$, e um troco W, o algoritmo guloso só tem garantia de correção caso A seja canônico. De toda forma, o algoritmo é simples de entender, eficiente em tempo de execução, tendo complexidade assintótica $O(m)$, e em memória, tendo complexidade

assintótica $O(1)$.

O funcionamento do algoritmo é bem simples: para um troco W, será escolhida a maior moeda j tal que $C_j \leq W$. Após essa moeda ser escolhida como parte da solução, o troco resultante W' passa a ser $W' = W - C_j$. Novamente, será escolhida a maior moeda k tal que $C_k \leq W'$, resultando em um novo troco $W'' = W' - C_k$. Este processo é repetido até que o valor do troco atual seja 0. A união de todas as moedas escolhidas durante o processo é a solução, a qual foi denominada como *representação gulosa*.

Um detalhe importante da implementação do algoritmo é a forma de fazer as escolhas da maior moeda j tal que $C_j \leq W$ para o troco W atual de forma eficiente. Se feita de forma inocente, o processo pode acabar tendo complexidade $O(W)$, como por exemplo: se $A = \{1\}$ e a cada iteração do algoritmo o troco W passa a ser $W' = W - 1$, então $O(W)$ iterações serão feitas.

Para melhorar tal estratégia, pode-se iterar pelo sistema de moedas de trás para frente e determinar quantas vezes cada moeda deve estar presente na solução final em $O(1)$; isto é, começando da maior moeda m, indo para a moeda $m-1$ e repetindo o processo até chegar à moeda 1. Quando na moeda i, com troco atual W', o algoritmo deve determinar a quantidade máxima E_i de moedas desse tipo que devem fazer parte da *representação gulosa*, onde E_i, nesse caso, pode ser representada por $E_i = \lfloor \frac{W'}{C_i} \rfloor$, onde $\lfloor \frac{a}{b} \rfloor$ representa a divisão inteira de a por b. Assim, determinada a quantidade E_i, o valor de W' é atualizado para $W'' = W' - E_i \cdot C_i$ e o algoritmo continua, de forma a processar a moeda $i-1$ no próximo passo. Como o algoritmo itera pelas m moedas, resolvendo cada passo em $O(1)$, a complexidade final em tempo de execução, confirmadamente, é $O(m) = O(m) \cdot O(1)$, onde o uso de memória é, claramente, $O(1)$.

O Algoritmo 4 descrito no parágrafo anterior pode ser visualizado a seguir:

Algoritmo 4: Problema do Troco - Algoritmo Guloso

```
1  input: a coin system A = {C_1, C_2, ..., C_m} and a non negative integer W
2      representing the change
3  output: a non negative integer representing |GUL_A(W)|
4
5  total ← 0
6  for i from m to 1, do:
7   q ← ⌊W / C_i⌋
8   total ← total + q
9   W ← W − qC_i
10
11 return total
```

Algoritmo de Programação Dinâmica para o Problema do Troco

Seguindo a estrutura para resolver um problema utilizando programação dinâmica, deve-se primeiramente caracterizar a estrutura de uma solução ótima. A definição do

problema do troco é: dado um sistema de moedas $A = \{C_1, C_2, ..., C_m\}$, com $C_1 = 1$, e um troco W, qual é a menor quantia de moedas presentes em A que seus valores somados resultam em W? Com isso em mente, uma função $F(W)$ pode ser definida com o seguinte significado: quantidade mínima de moedas pertecentes a A que somam W.

Assim, dado um estado (W) em F, a única opção de transição é escolher uma moeda i dentre as m tal que $C_i \leq W$ e verificar o que acontece com seu estado posterior: o estado $(W - C_i)$. A melhor opção entre as moedas candidatas é o valor ótimo para $F(W)$, já que os seus subproblemas compartilham a sua mesma estrutura. Assim, a recorrência para o problema do troco pode ser definida da seguinte forma:

$$F(W) = \begin{cases} 0, & \text{se } W = 0 \\ 1 + \min\{F(W - C_i) | 1 \leq i \leq m, C_i \leq W\}, & \text{caso contrário} \end{cases} \tag{1.8}$$

Para concretizar a recorrência dita acima, o Algoritmo 5 que utiliza a estratégia *top down* é mostrado a seguir:

Algoritmo 5: Problema do Troco - Programação Dinâmica

```
1  function: F(A, W)
2  input: a coin system A = {C_1, C_2, ..., C_m} and
3      a non negative integer W representing the change
4  output: a non negative integer representing |OPT_A(W)|
5
6  if W = 0, then:
7   return 0
8
9  if state (A,W) is already calculated, then:
10  return answer[W]
11
12 total ← ∞
13 for i from 1 to m, do:
14  if C_i ≤ W, then:
15   total ← min(total, F(A, W − C_i) + 1)
16
17 answer[W] = total
18 return total
```

Como existem W estados possíveis em F e cada um é calculado apenas uma vez em $O(W)$, a complexidade final do algoritmo em tempo de execução é $O(W \cdot m)$, enquanto a complexidade em memória é $O(m)$, devido à tabela utilizada para memorizar os valores já calculados. Vale notar que essa versão da programação dinâmica faz menos uso de memória do que a versão apresentada na Seção 1.3.3, por conta da troca de responsabilidade de escolher a moeda a ser escolhida ser passada para a transição.

Algoritmo de Determinação de Canonicidade para o Problema do Troco

Como dito anteriormente, é possível determinar se um dado sistemas de moedas é canônico em $O(m^2)$ (onde m é a quantidade de moedas do sistema) utilizando um algoritmo de canonicidade proposto por Cai (2009). Seu funcionamento pode ser visualizado no Algoritmo 6:

Algoritmo 6: Problema do Troco - Algoritmo de Canonicidade

```
1  input: a coin system A = {C_1, C_2, ..., C_m}, with m ≥ 3
2  output: a state representing whether A is canonical
3
4  q ← ⌊C_3 / C_2⌋
5  r ← C_3 mod C_2
6  if 0 < r < C_2 − q, then:
7   return A is not canonical
8
9  for i from m − 1 to 1, do:
10  for j from i to 1, do:
11   if C_i + C_j > C_m and |GUL_A(C_i + C_j)| > 2, then:
12    return A is not canonical
13
14 return A is canonical
```

Como pode ser visto no trabalho de Cai (2009), se algum par de moedas C_i e C_j do sistema for maior que a maior moeda (C_m) do sistema e a solução gulosa dada para sua soma não for ótima, então a base é determinada como não canônica.

Se $C_i + C_j > C_m$, é fácil ver que $|OPT_A(C_i + C_j)| = 2$, pois escolher C_i e C_j é a melhor opção. Assim, se $|GUL_A(C_i + C_j)| > 2$, então o sistema A é considerado não canônico. O algoritmo guloso com certeza escolherá a moeda $m+1$ em sua primeira etapa, resultando em um troco de valor $W = C_i + C_j - C_m$. Se uma moeda k tal que $C_k = W$ não existir em A, então serão necessárias pelo menos 2 moedas para alcançar o valor W, o que implicaria que $|GUL_A(C_i + C_j)| > 2$, já que 1 moeda foi escolhida inicialmente. Assim, resta determinar se um valor inteiro ($W = C_i + C_j - C_m$, neste caso) existe no sistema A. Essa tarefa pode ser facilmente resolvida com uma tabela *hash*: um pré-processamento de inserção de todos as moedas de A na tabela *hash* pode ser feito antes de qualquer etapa do algoritmo, possibilitando a consulta de elementos em $O(1)$.

2 Metodologia

Este capítulo tem como objetivo descrever como os experimentos foram realizados na Seção 2.3, além das ferramentas na Seção 2.2 e fontes utilizadas para a realização dos mesmos na Seção 2.1.

2.1 Revisão da Literatura

Para desenvolvimento deste trabalho, duas fontes principais de informação e dados foram utilizadas: artigos e livros. Para a busca, acesso e leitura de artigos foi utilizado o Portal de Periódicos Capes[1], onde os alunos da Universidade de Brasília podem acessar uma vasta quantidade de artigos de qualidade utilizando seus *e-mails* acadêmicos e IPs dentro da própria universidade. Um dos livros (CORMEN, 2011) foi consultado por meio de compra e os livros restantes foram consultados na Biblioteca do Gama e na Biblioteca Central da UnB, localizada no campus Darcy Ribeiro.

2.2 Ferramentas

Foi utilizada a linguagem de programação C++, seguindo o padrão C++14, com utilização do compilador gcc versão 5.4.0, para realizar a implementação dos algoritmos descritos nesse trabalho e os cálculos dos tempos de execução de cada um deles. Além disso, a linguagem de programação Python em sua versão 3.5.2, juntamente com a biblioteca matplotlib versão 3.0.2, foi utilizada para a renderização dos gráficos. Ubuntu 16.04.1 foi o sistema operacional em que os experimentos foram realizados.

As implementações dos algoritmos foram executadas em um *notebook* LENOVO modelo 80YH com o processador i7-7500U 2.70GHz da Intel e 4 GB de memória RAM.

2.3 Realização dos Experimentos

2.3.1 Motivação

Como exposto anteriormente neste trabalho, o algoritmo guloso para resolução do problema do troco tem vantagens tanto em complexidade de tempo de execução quanto em complexidade de memória, apesar da não garantia de corretude em todos os casos. Caso a canonicidade de um sistema de moedas fosse conhecida de antemão, a aplicação do algoritmo guloso traria vantagem em todos os aspectos, já que, por definição, a corretude

[1] http://www.periodicos.capes.gov.br/

seria garantida. Assim, surgem dois cenários possíveis para resolver o problema do troco para um sistema de moedas $A = \{C_1, C_2, ..., C_m\}$ e um troco W dado quaisquer:

1. Determinar a canonicidade do sistema de moedas A em $O(m^2)$ com o Algoritmo de Canonicidade descrito na Seção 1.3.3.

   1.1. Caso A seja canônico, obter a resposta em $O(m)$ com o Algoritmo Guloso descrito na Seção 1.3.3.

   1.2. Caso A não seja canônico, obter a resposta em $O(Wm)$ com o Algoritmo de Programação Dinâmica descrito na Seção 1.3.3.

2. Obter a resposta em $O(Wm)$ com o Algoritmo de Programação Dinâmica descrito na Seção 1.3.3.

É possível verificar que a complexidade do tempo de execução do cenário 1 é menor que a do cenário 2 quando W é maior que m, pois $W \in o(m)$ e, portanto, $Wm \in o(m^2)$, e, também, o sistema de moedas A é canônico. Caso contrário, o cenário 2, sem qualquer tipo de verificação da canonicidade de A, acaba se mostrando melhor, já que, no pior caso, o cenário 2 é um subconjunto do cenário 1.

Levando em conta as observações anteriores, foram realizados experimentos aleatórios com sistemas de moedas para comparar os tempos de execução do Algoritmo Guloso, este em conjunto com o Algoritmo de Canonicidade, com o Algoritmo de Programação Dinâmica. A unidade de tempo de todos os tempos de execução descritos neste trabalho é o milissegundo.

2.3.2 Métodos de Geração de Sistemas de Moeda Randômicos

Os experimentos a seguir utilizam conjuntos diferentes de sistemas de moeda: os não canônicos, os não necessariamente canônicos (podendo ser canônicos ou não canônicos) e os canônicos. Para cada um desses conjuntos, será descrito brevemente como seu método de geração foi determinado.

Sistemas Não Canônicos

Supondo que queiramos sortear um sistema de moedas A não canônico com $N > 1$ moedas, os passos para gerá-lo são:

1. Inserir o elemento 1 em A.

2. Sortear $N - 1$ inteiros distintos pseudo-aleatórios, sendo todos diferentes de 1, e inseri-los em A.

3. Verificar se A é canônico segundo o Algoritmo de Canonicidade descrito na Seção 1.3.3: caso não seja, finalizar o processo. Caso contrário, retirar todos os valores de A e voltar ao passo 1.

Sistemas Não Necessariamente Canônicos

Supondo que queiramos sortear um sistema de moedas A não necessariamente canônico com $N > 1$ moedas, os passos para gerá-lo são:

1. Inserir o elemento 1 em A.
2. Sortear $N-1$ inteiros distintos pseudo-aleatórios, sendo todos diferentes de 1, inseríos em A e finalizar o processo.

Sistemas Canônicos

Um método de geração similar ao dos sistemas não canônicos seria um bom candidato de método de geração para os sistemas canônicos. Acontece que escolher N valores randomicamente de tal forma que eles formem um sistema canônico acontece muito raramente para tamanhos relativamente grandes de N ($N \geq 50$, no caso). Dessa forma, para tamanhos pequenos (até 25), os sistemas canônicos foram gerados com um método similar aos dos sistemas não canônicos:

1. Inserir o elemento 1 em A.
2. Sortear $N-1$ inteiros distintos pseudo-aleatórios, sendo todos diferentes de 1, e inseríos em A.
3. Verificar se A é canônico segundo o Algoritmo de Canonicidade descrito na Seção 1.3.3: caso seja, finalizar o processo. Caso contrário, retirar todos os valores de A e voltar ao passo 1.

Para sistemas de moedas de maior tamanho (50 ou mais), foram pré-processados sistemas de moedas canônicos que são de fácil geração, como por exemplo: sistemas compostos por potências de um número p qualquer e sistemas compostos por múltiplos de um número p qualquer. Dentre esses sistemas pré-processados, para cada um dos tamanhos de sistemas utilizados neste trabalho, 100 deles são sorteados para a realização de cada instância dos experimentos, com os detalhes de cada um sendo descritos na seção a seguir.

2.3.3 Experimentos

Foram feitos 3 experimentos explorando sistemas de moedas randômicos (canônicos ou não). Cada um dos 3 experimentos será detalhado nos tópicos seguintes.

Experimento 1 - Tempo de execução: Guloso + Canonicidade vs. Programação Dinâmica, com sistemas de moedas canônicos

O experimento foi dividido em quatro passos, listados a seguir:

1. Foi definido um conjunto T de tamanhos de sistemas de moedas a serem testados.

2. Para cada valor t em T, foram sorteados N sistemas de moedas canônicos randômicos, com base no gerador descrito na Seção 2.3.2, $A_{t,i} = \{C_1, C_2, ..., C_t\}$ e Q valores de troco randômicos $q_{t,i}$.

3. Para cada sistema de moedas randômico $A_{t,i}$, foram calculados dois tempos de execução:

   3.1. $D_{t,i} = \sum_{n=1}^{Q} T_D(A_{t,i}, q_n)$, onde $T_D(x, y)$ representa o tempo de execução para resolver o problema para o troco y em relação ao sistema de moedas x utilizando o Algoritmo de Programação Dinâmica descrito na Seção 1.3.3.

   3.2. $G_{t,i} = \sum_{n=1}^{Q} T_G(A_{t,i}, q_n) + T_C(A_i)$, onde $T_G(x, y)$ representa o tempo de execução para resolver o problema para o troco y em relação ao sistema de moedas x utilizando o Algoritmo Guloso descrito na Seção 1.3.3 e $T_C(k)$ representa o tempo de execução para determinar se o sistema de moedas k é canônico usando o Algoritmo de Canonicidade descrito na Seção 1.3.3.

4. Por fim, para cada valor t em T, foram calculados dois valores:

   4.1. $\bar{D}_t = \frac{1}{N} \sum_{n=1}^{N} D_{t,n}$ - representando a média dos tempos de execução do Algoritmo de Programação Dinâmica descrito na Seção 1.3.3 para cada um dos sistemas de moedas randômicos A_i selecionados para o dado tamanho t.

   4.2. $\bar{G}_t = \frac{1}{N} \sum_{n=1}^{N} G_{t,n}$ - representando a média dos tempos de execução do Algoritmo Guloso descrito na Seção 1.3.3 (com a adição de uma única execução do Algoritmo de Canonicidade descrito na Seção 1.3.3) para cada um dos sistemas de moedas randômicos A_i selecionados para o dado tamanho t.

Experimento 2 - Tempo de execução: Guloso + Canonicidade vs. Programação Dinâmica, com sistemas de moedas não necessariamente canônicos

O experimento 2 segue os mesmos passos do experimento 1, com a única diferença de que os sistemas de moedas sorteados randomicamente foram sorteados com base no gerador descrito na Seção 2.3.2.

Experimento 3 - Taxa média de acertos do Algoritmo Guloso em sistemas de moedas não canônicos

O experimento foi dividido em quatro passos, listados a seguir:

1. Foi definido um conjunto T de tamanhos de sistemas de moedas a serem testados.

2. Para cada valor t em T, foram sorteados N sistemas de moedas não canônicos randômicos, com base no gerador descrito na Seção 2.3.2, $A_{t,i} = \{C_1, C_2, ..., C_t\}$ e Q valores de troco randômicos $q_{t,i}$.

3. Para cada sistema de moedas não canônico randômico $A_{t,i}$, foi calculado um valor representando a taxa média de acertos[2] do Algoritmo Guloso descrito na Seção 1.3.3 para cada uma das Q consultas sorteadas:

   3.1. $K_{t,i} = \frac{|O|}{Q}$, onde $O = \{i | 1 \leq i \leq Q, |GUL_{A_{t,i}}(q_{t,i})| = |OPT_{A_{t,i}}(q_{t,i})|\}$.

4. Por fim, para cada valor t em T, foi calculado um valor representando a média das taxas médias de cada uma dos sistemas de moedas com forma $A_{t,i}$:

   4.1. $\bar{K_t} = \frac{1}{N} \sum_{i=1}^{N} K_{t,i}$.

[2] Um acerto acontece, para um sistema A não canônico e um troco W, quando $|GUL_A(W)| = |OPT_A(W)|$

3 Resultados e Discussões

Neste capítulo serão apresentados os resultados e as discussões para cada um dos 3 experimentos descritos na Seção 2.3. As Seções 3.1, 3.2, e 3.3 apresentam, respectivamente, os resultados e as discussões sobre os experimentos 1, 2 e 3. Em cada uma dessas seções, as nomenclaturas Algoritmo Guloso, Algoritmo de Programação Dinâmica e Algoritmo de Canonicidade estarão se referenciando aos algoritmos descritos na Seção 1.3.3.

3.1 Experimento 1

Como descrito no item 1 da Seção 2.3.3, um conjunto T de tamanhos de sistemas de moedas a serem testados foi escolhido para a realização deste experimento: neste caso, $T = \{5, 10, 25, 50, 100, 250, 500, 1000\}$. Além disso, os valores de N, Q, C_t e $q_{t,i}$, descritos no item 2 da mesma seção, adotados neste experimento foram: $N = 100$, $Q = 1000$, $1 < C_t \leq 10^4$ e $0 \leq q_{t,i} \leq 10^4$. A Figura 1 apresenta os resultados para o Experimento 1 em forma de gráfico, onde a linha azul representa a variação da variável $\bar{G}_t$ e a linha laranja representa a variação da variável $\bar{D}_t$, ambas descritas no item 4 da Seção 2.3.3:

É possível notar que a soma dos tempos de execução do Algoritmo Guloso para resolver cada uma das Q consultas com o tempo de execução do Algoritmo de Canonicidade é superior ao tempo de execução do Algoritmo de Programação Dinâmica até os sistemas de moedas com tamanho 76, valor este obtido por meio de interpolação linear.

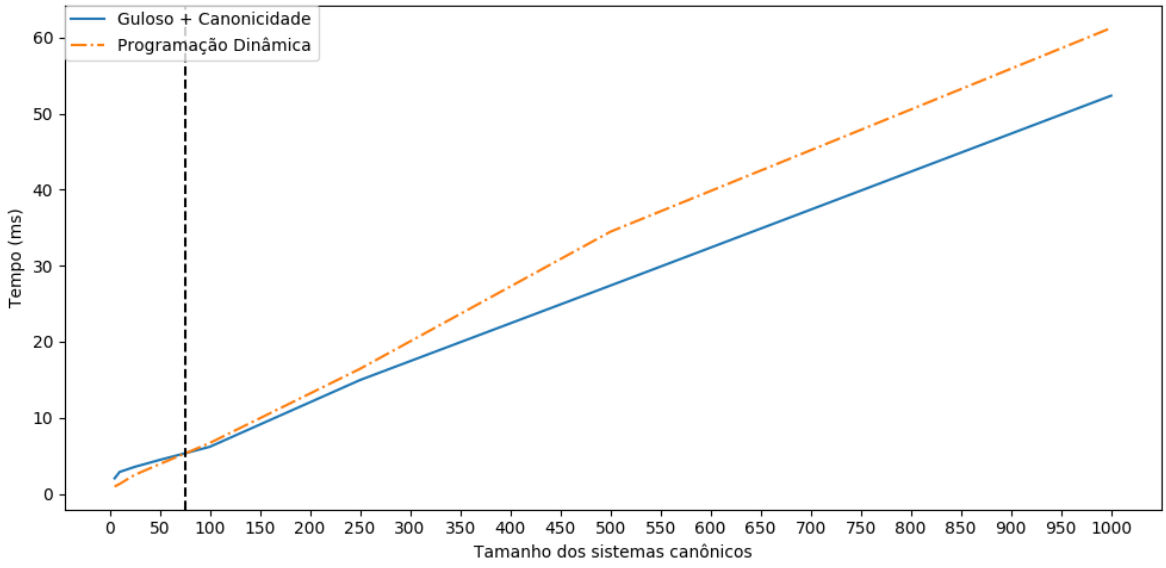

Figura 1 – Gráficos dos tempos de execução dos algoritmos em cima de sistemas de moedas canônicos

A partir deste ponto, o Algoritmo de Programação Dinâmica se mostra mais ineficiente.

As complexidades assintóticas de tempo de execução para o cenário Guloso + Canonicidade é $O(Qm + m^2)$ e para o cenário Programação Dinâmica, $O(m \cdot q_{t,i})$, onde m é o tamanho do sistema de moedas. É importante notar que o número de consultas Q não influencia diretamente na complexidade do Algoritmo de Programação Dinâmica pois a memoização garante a complexidade apresentada de forma amortizada.

Levando em conta essas complexidades, esperava-se, teoricamente, que os tempos de execução do Algoritmo de Programação Dinâmica sempre fossem mais elevados do que os do cenário Guloso + Canonicidade. Acontece que o fatores Q e m são relativamente próximos do maior valor possível de $q_{t,i}$, possibilitando que, para tamanhos pequenos, diferenças pequenas como as de implementação tragam um resultado inesperado. Conforme o tamanho cresce, a diferença de constante entre as complexidades é potencializada pelo fator $\frac{q_{t_i}}{(m+Q)} = 5$, fazendo com que essas pequenas diferenças, provavelmente com custos constantes, não influenciem nos tempos dadas as ordens de grandeza relativamente elevadas, o que acarreta em resultados esperados.

Com as observações feitas acima, pode-se fazer algumas afirmações:

- Dado um sistema de moedas com tamanho até 76, é vantajoso utilizar o Algoritmo de Programação Dinâmica em relação à combinação Guloso + Canonicidade para resolver o problema do troco.

- Para tamanhos maiores que 76, pode ser vantajoso tentar utilizar a combinação Guloso + Canonicidade antes de já resolver o problema com o Algoritmo de Programação Dinâmica. Isso depende da probabilidade de um dado sistema de moedas ser canônico ou não. Caso a probabilidade do sistema ser canônico seja relativamente alta, então o valor esperado de tempo de execução total pode ser menor caso a combinação Guloso + Canonicidade seja utilizada previamente na tentativa de resolver o problema do troco. Caso contrário, utilizar o Algoritmo de Programação Dinâmica diretamente pode ser mais vantajoso.

3.2 Experimento 2

Os valores para este experimento são os mesmos que os definidos na Seção 3.1. A única diferença deste experimento para o Experimento 1 é a utilização dos sistemas de moedas randômicos: aqui, eles não necessariamente são canônicos. A Figura 2 apresenta os resultados para o Experimento 2 em forma de gráfico:

É notável que a linha laranja, representando os tempos de execução do Algoritmo de Programação Dinâmica, se comporta de maneira semelhante à linha laranja do experimento 1, apresenta na Figura 1. Em contrapartida, a linha azul, representando a

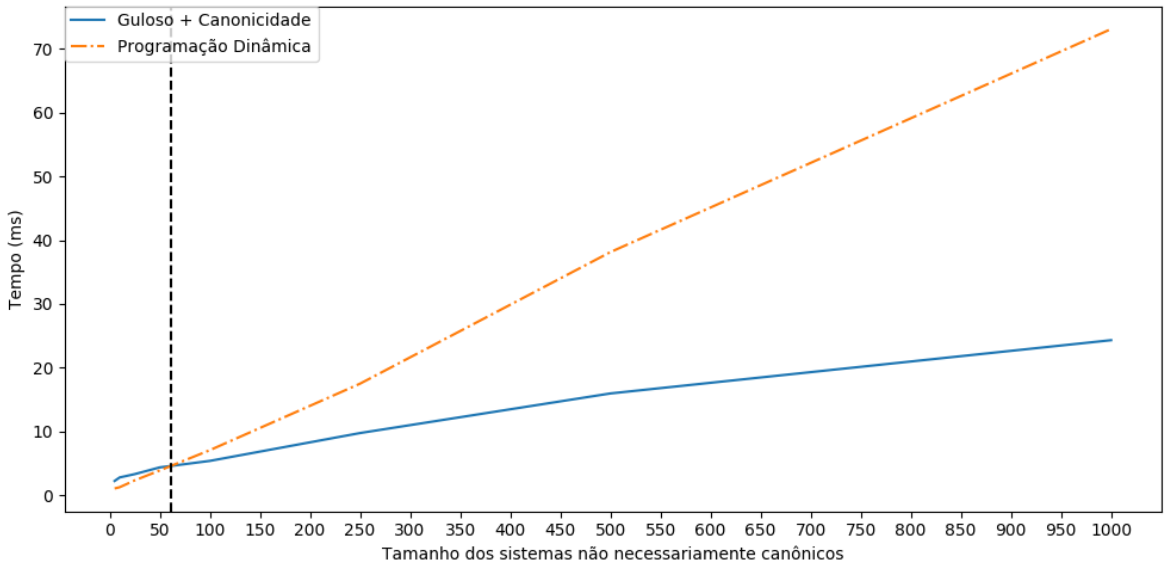

Figura 2 – Gráficos dos tempos de execução dos algoritmos em cima de sistemas de moedas não necessariamente canônicos

combinação Guloso + Canonicidade, apresenta uma taxa de crescimento menor quando comparada com a linha azul do experimento 1.

A primeira observação que pode ser feita é: dada a diferença das inclinações das linhas azuis dos dois experimentos, pode-se imaginar que a distribuição de sistemas de moedas canônicos em relação a sistemas de moedas não canônicos é baixa. Em outras palavras, ao sortear um sistema de moedas pseudo-aleatoriamente, é mais provável que o mesmo seja não canônico. Um possível argumento para tal afirmação é: se os sistemas canônicos fossem mais prováveis, então em um sorteio de sistemas não necessariamente canônicos, o comportamento dos tempos de execução deveria ser parecido com o comportamento de um ensaio composto somente por sistemas canônicos, como feito no experimento 1.

Supondo que a hipótese apresentada acima esteja correta, outra observação pode ser feita: a canonicidade de um dado sistema não influencia os tempos de execução do Algoritmo de Programação Dinâmica tanto quanto influencia os tempos de execução da combinação Guloso + Canonicidade. Para melhor visualização das hipóteses e observações, visualize a Figura 3:

Supondo também que a hipótese da baixa probabilidade de que sistemas randômicos sejam canônicos seja verdadeira, então uma pergunta interessante pode ser feita: por que os tempos de execução para os sistemas canônicos (utilizados no Experimento 1) utilizando a combinação Guloso + Canonicidade são maiores? Uma possível explicação é: o Algoritmo 6 descrito na Seção 1.3.3 faz verificações para todo par de moedas presente no sistema de moedas, e assim que um par determina a não canonicidade do sistema, o algoritmo é encerrado. No caso dos sistemas canônicos, nenhum par determinará a não

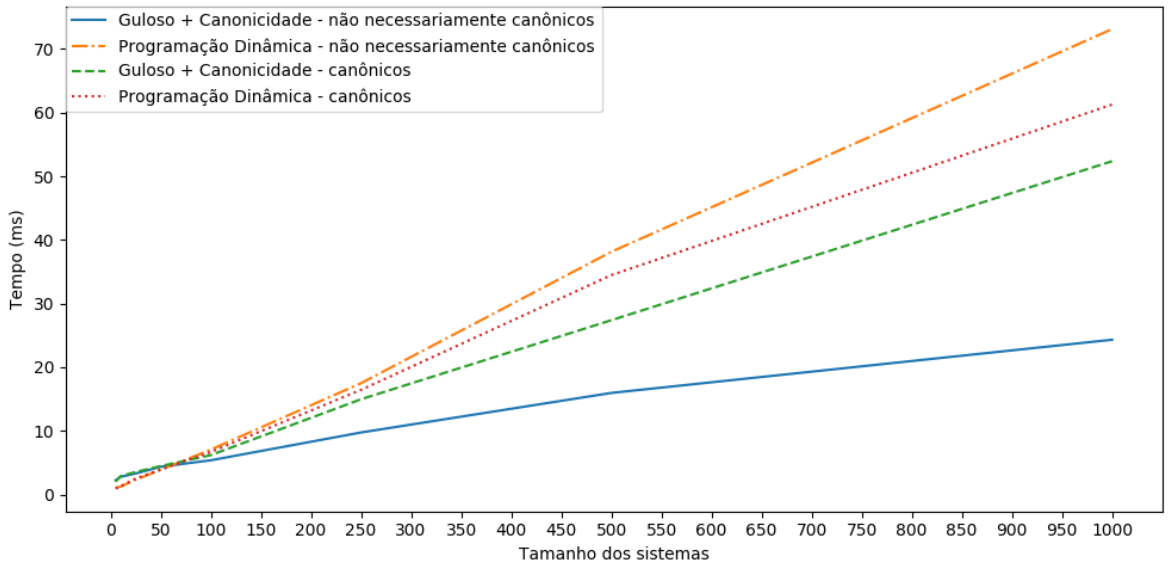

Figura 3 – Gráficos dos tempos de execução dos algoritmos em cima de sistemas de moedas

canonicidade do sistema, fazendo com que todos os pares sejam visitados. No caso dos sistemas não canônicos, a quantidade média de pares visitados pode ser relativamente baixa em relação aos m^2 pares possíveis, resultando no menor tempo de execução por parte do algoritmo em cima de sistemas de moedas não necessariamente canônicos.

Vale lembrar que, neste experimento, a combinação de Guloso + Canonicidade é heurística. Como as canonicidades dos sistemas não são garantidas, os resultados gerados podem não ser ótimos.

3.3 Experimento 3

Como descrito no item 1 da Seção 2.3.3, um conjunto T de tamanhos de sistemas de moedas a serem testados foi escolhido para a realização deste experimento: neste caso, $T = \{5, 10, 25, 50, 100, 250, 500, 1000\}$. Além disso, os valores de N, Q, C_t e $q_{t,i}$, descritos no item 2 da mesma seção, adotados neste experimento foram: $N = 100$, $Q = 1000$, $1 < C_t \leq 10^4$ e $0 \leq q_{t,i} \leq 10^4$. A Figura 4 apresenta os resultados para o Experimento 3 em forma de gráfico, onde a linha azul representa a variação da variável $\bar{K}_t$ descrita no item 4 da Seção 2.3.3:

Um acerto neste experimento acontece quando, para um troco W qualquer e um sistema de moedas não canônico, o Algoritmo Guloso retorna a mesma resposta que o Algoritmo de Programação Dinâmica.

É possível perceber que há uma correlação entre a taxa média de acertos e o tamanho do sistema de moedas: quanto maior o sistema de moedas, mais improvável que

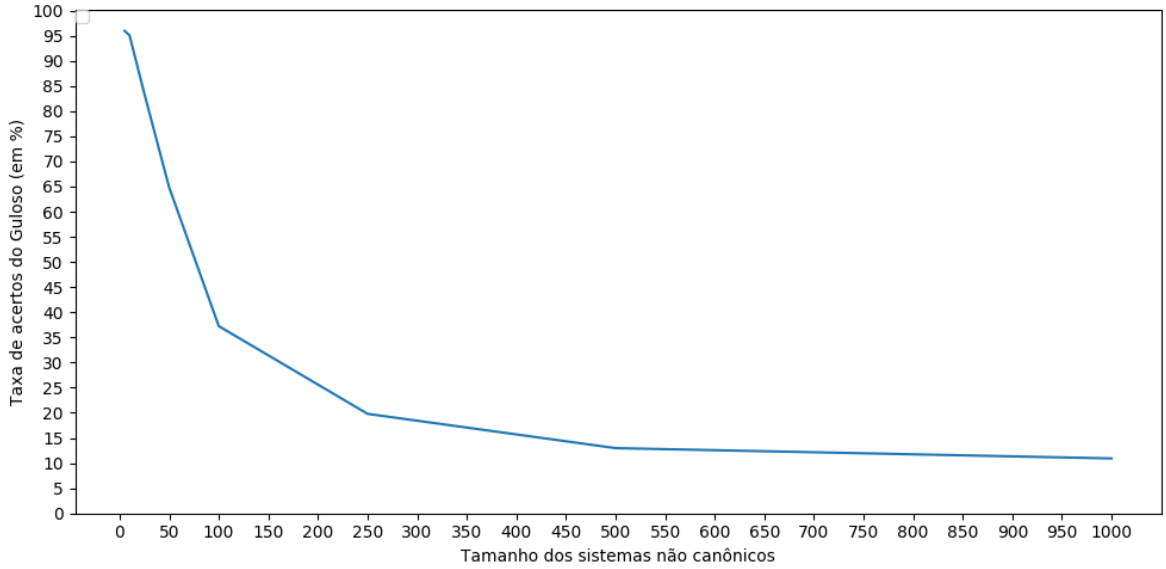

Figura 4 – Gráfico das taxas médias de acertos do Algoritmo Guloso em sistemas de moedas não canônicos

um acerto ocorra. O decaímento da taxa é bem elevado na transição dos tamanhos iniciais, decrescendo conforme os tamanhos aumentam.

Esse gráfico nos permite chegar a uma conclusão: utilizando o Algoritmo Guloso, a chance de não se chegar em uma solução ótima é maior que 50% para sistemas de tamanhos maiores ou iguais a 100. Em um cenário onde as taxas médias de acertos fossem muito elevadas (como é o caso de sistemas de moedas de tamanho 5, onde a taxa é 95%), a utilização do Algoritmo Guloso de forma direta poderia ser uma ótima estratégia para resolver o problema do troco ou até mesmo para utilização do valor retornado como heurística, dada a alta probabilidade de acerto.

4 Considerações Finais

Neste trabalho, foram apresentados algoritmos para a resolução do problema do troco e para a determinação de canonicidade de um sistema de moedas, assim como a análise dos seus tempos de execução. Foi visto que, na prática, a combinação do Algoritmo Guloso com o Algoritmo de Canonicidade é mais rápida do que o Algoritmo de Programação Dinâmica para a maioria dos tamanhos de sistema de moedas, com exceção de tamanhos pequenos.

Foi verificado que a geração de sistema de moedas canônicos pseudo-aleatórios é uma tarefa complicada. Para tamanhos de sistemas maiores ou iguais a 50, a quantidade de sistemas canônicos parece ser muito inferior à quantidade de sistemas não canônicos. Geradores com certos padrões foram utilizados para a geração de sistemas canônicos, mas seus usos restringem o espaço e acarretam em uma gama muito menor de amostras no conjunto a ser testado.

Além disso, foi possível perceber que a taxa de acerto do algoritmo guloso para sistemas de moedas randômicos é de quase 20% para sistemas de moedas com tamanhos a partir de 100, mas bem elevada (em torno de 90%) para sistemas relativamente pequenos (em torno de 5 moedas).

Trabalhos Futuros

Levando em conta as discussões apresentadas no Capítulo 3, a lista a seguir é uma sugestão de trabalhos futuros que podem acarretar resultados interessantes em complemento aos deste trabalho:

- estudo de distribuições de probabilidade de sistemas de moedas canônicos diante de um conjunto de sistemas de moedas randômicos;
- investigar se há formas de redução dos tempos de execução da combinação do Algoritmo Guloso com o Algoritmo de Canonicidade, para que os mesmos sempre sejam mais rápidos que o Algoritmo de Programação Dinâmica, mesmo para sistemas de moedas pequenos;
- implementar os algoritmos apresentados neste trabalho em outras linguagens de programação e mensurar e comparar os tempos de execução dos mesmos;
- estudar formas de geração de sistemas de moedas canônicos mais eficientes do que as totalmente pseudo-randômicas e diferentes das gerações de sistemas formados por múltiplos e potências.

Referências

ANDONOV, R.; POIRRIEZ, V.; RAJOPADHYE, S. Unbounded knapsack problem: Dynamic programming revisited. 2000. Citado na página 23.

CAI, X. Canonical coin systems for change-making problems. Department of Computer Science and Engineering, Shanghai Jiao Tong University, 2009. Disponível em: <https://arxiv.org/pdf/0809.0400.pdf>. Citado 3 vezes nas páginas 15, 26 e 29.

CORMEN, T. *Introduction to Algorithms*. 3rd. ed. [S.l.]: MIT Press, 2011. Citado 3 vezes nas páginas 19, 22 e 31.

DIJKSTRA, E. A short introduction to the art of programming. 1971. Disponível em: <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD316.html>. Citado na página 17.

HOFFMAN, K.; RALPHS, T. Integer and combinatorial optimization. LIMAV, Université de Valenciennes Le Mont Houy, 2012. Disponível em: <https://www.sciencedirect.com/science/article/abs/pii/S0377221799002659>. Citado na página 24.

JENKINS, P. Greedy algorithms and the making change problem. Department of Computer Science, University of Warwick, 2004. Disponível em: <https://warwick.ac.uk/fac/sci/dcs/research/em/publications/web-em/01/greedy.pdf>. Citado na página 26.

KLEINBERG, J.; TARDOS, E. *Algorithm Design*. 2nd. ed. [S.l.]: Pearson Education, 2011. Citado na página 26.

KNUTH, D. *The Art of Computer Programming Vol. 1*. Boston, MA, USA: Addison Wesley Longman, 1968. Citado 2 vezes nas páginas 17 e 19.

NIEWIAROWSKA, A.; ADAMASZEK, M. Combinatorics of the change-making problem. European Journal of Combinatorics, 2010. Disponível em: <https://arxiv.org/PS_cache/arxiv/pdf/0801/0801.0120v2.pdf>. Citado na página 26.

PEARL, J. Heuristics: Intelligent search strategies for computer problem solving. Department of Computer Science, Cornell University, 1984. Citado na página 17.

Apêndices

APÊNDICE A – Algoritmo de Programação Dinâmica para o Problema do Troco

```cpp
template <typename T>
class CoinSystem {
public:
  CoinSystem() {}

  CoinSystem(const std::initializer_list<T> l) {
    base.assign(l);
  }

  void insert(const T coin) {
    base.push_back(coin);
    std::sort(base.begin(), base.end());
    base.erase(std::unique(base.begin(), base.end()), base.end());
  }

  Representation<T> optimal_solution(const T change) {
    if(change < 0) {
      throw std::invalid_argument("The change should be a non negative integer.");
    }

    std::vector<T> memo(change + 1), parent(change + 1, -1);
    dynamic_solve(change, memo, parent);
    Representation<T> E = construct_solution(change, parent);

    return E;
  }

private:
  std::vector<T> base;

  T dynamic_solve(T change, std::vector<T> &memo, std::vector<T> &parent) {
    if(change == 0) {
      return 0;
    }

    T &answer = memo[change];
    if(~answer) {
      return answer;
```

```
39      }
40
41      answer = std::numeric_limits<T>::max();
42      for(int i = 0; i < (int) base.size(); i++) {
43        if(change >= base[i]) {
44          T current = dynamic_solve(change - base[i], memo, parent) + 1;
45          if(current < answer) {
46            answer = current;
47            parent[change] = i;
48          }
49        }
50      }
51
52      return answer;
53    }
54
55    Representation<T> construct_solution(T change, const std::vector<T> &
       parent) const {
56      Representation<T> E((int) base.size());
57
58      while(change > 0) {
59        T chosen_index = parent[change];
60        E.add(chosen_index, 1);
61        T chosen_coin = base[chosen_index];
62        T next_change = change - chosen_coin;
63        change = next_change;
64      }
65
66      return E;
67    }
68  };
```

APÊNDICE B – Algoritmo Guloso para o Problema do Troco

```cpp
template <typename T>
class CoinSystem {
public:
  CoinSystem() {}

  CoinSystem(const std::initializer_list<T> l) {
    base.assign(l);
  }

  void insert(const T coin) {
    base.push_back(coin);
    std::sort(base.begin(), base.end());
    base.erase(std::unique(base.begin(), base.end()), base.end());
  }

  Representation<T> greedy_solution(T change) const {
    if(change < 0) {
      throw std::invalid_argument("The change should be a non negative integer.");
    }

    Representation<T> E((int) base.size());
    for(int i = (int) base.size() - 1; i >= 0; i--) {
      T q = change / base[i];
      E.add(i, q);
      change -= q * base[i];
    }

    return E;
  }

private:
  std::vector<T> base;
};
```

APÊNDICE C – Algoritmo de Canonicidade para o Problema do Troco

```
template <typename T>
class CoinSystem {
public:
  CoinSystem() {}

  CoinSystem(const std::initializer_list<T> l) {
    base.assign(l);
  }

  void insert(const T coin) {
    base.push_back(coin);
    std::sort(base.begin(), base.end());
    base.erase(std::unique(base.begin(), base.end()), base.end());
  }

  bool is_canonical() const {
    if((int) base.size() < 0) {
      throw std::domain_error("The number of coins in the Coin System should be at least 1.");
    }

    if((int) base.size() <= 2) {
      return true;
    }

    T q = base[2] / base[1];
    T r = base[2] % base[1];
    if(0 < r and r < base[2] - q) {
      return false;
    }

    std::unordered_set<T> values_in(base.begin(), base.end());

    int m = (int) base.size() - 1;
    for(int i = m - 1; i >= 0; i--) {
      for(int j = i; j >= 0; j--) {
        if(base[i] + base[j] > base.back() and !values_in.count(base[i] + base[j] - base.back())) {
          return false;
        }
```

```
39        }
40      }
41
42      return true;
43    }
44
45  private:
46    std::vector<T> base;
47  };
```

APÊNDICE D – Função de Randomização

```
#include <random>
namespace MtRandomizer {
  int rand(const int a, const int b) {
    static std::seed_seq seq {
      (uint64_t) std::chrono::duration_cast<std::chrono::nanoseconds>(
    std::chrono::high_resolution_clock::now().time_since_epoch()).count()
    ,
      (uint64_t) __builtin_ia32_rdtsc()
    };
    static std::mt19937 mt(seq);
    return std::uniform_int_distribution<int>(a, b)(mt);
  }
};
```

APÊNDICE E – Benchmark para uma função qualquer

```
#include <chrono>
namespace Benchmark {
  template <typename F>
  double benchmark_event(F &&f) {
    using target_duration = std::chrono::duration<double, std::ratio<1,
    1>>;

    auto start = std::chrono::steady_clock::now();
    f();
    auto end = std::chrono::steady_clock::now();

    return std::chrono::duration_cast<target_duration>(end - start).
    count();
  }
}
```