Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Estudo e Implementação da Árvore de Semigrupos Numéricos

Autor: Bruno Henrique Sousa Duarte
Orientador: Prof. Dr. Matheus Bernardini de Souza

Brasília, DF
2023

Bruno Henrique Sousa Duarte

Estudo e Implementação da Árvore de Semigrupos Numéricos

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. Matheus Bernardini de Souza

Brasília, DF

2023

Bruno Henrique Sousa Duarte

Estudo e Implementação da Árvore de Semigrupos Numéricos/ Bruno Henrique Sousa Duarte. – Brasília, DF, 2023-

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

Orientador: Prof. Dr. Matheus Bernardini de Souza

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2023.

1. Palavra-chave01. 2. Palavra-chave02. I. Prof. Dr. Matheus Bernardini de Souza. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Estudo e Implementação da Árvore de Semigrupos Numéricos

CDU 02:141:005.6

Bruno Henrique Sousa Duarte

Estudo e Implementação da Árvore de Semigrupos Numéricos

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.

Prof. Dr. Matheus Bernardini de Souza
Orientador

Prof. Dr. Victor do Nascimento Martins
Convidado 1

Prof. Dr. Edson Alves da Costa Júnior
Convidado 2

Brasília, DF
2023

Agradecimentos

Primeiramente agradeço a Deus pelo dom da vida e por ter me sustentado e guiado em minha jornada. Sou extremamente grato principalmente ao professor Matheus Bernardini por ter me concedido a honra e privilégio de ser meu orientador e pelas resenhas nas reuniões.

Agradecer meus pais, avós e familiares pelo apoio incondicional que me permitiu seguir em frente nos estudos e aos meus amigos, irmãos e parceiros da vida.

Por fim, gostaria de dedicar este trabalho especialmente aos meus amados avós, que mesmo que eles não estejam mais presentes fisicamente, confiaram a mim a continuidade de seus sonhos.

Resumo

Este trabalho refere-se ao desenvolvimento de conexões entre os conceitos computacionais e a área de estudo matemático de semigrupos numéricos, mais especificamente visando compreender a contagem de semigrupos numéricos por gênero dado. Abordaremos aqui as operações e técnicas necessárias para obter elementos da sequência (n_g) de forma rápida e com maior eficiência mediante as linguagens de programação. Desta forma, busca-se, com auxílio de estudos já elaborados anteriormente, definir quais as melhores alternativas computacionais tanto em função do tempo e execução das operações quanto em relação à particularidade das linguagens de programação. A construção da árvore de semigrupos numéricos foi implementada nas linguagens de programação Python e C++. Adicionalmente, foi empregado o Cilk++ para a paralelização do processo. Importante citar que para tal pesquisa adotou-se como referencial matemático os conceitos de árvore de semigrupos numéricos. Em complemento a isso, o presente estudo relaciona-se com as metodologias criadas por Fromentin e Hivert a fim de estabelecer as relações computacionais e matemáticas objetivadas. O conhecimento prévio, tanto de algumas estruturas de algoritmos quanto de alguns conceitos matemáticos colabora para a leitura e entendimento da presente pesquisa. Este trabalho obteve como resultado a contagem da sequência até n_{64}, juntamente com a criação de uma tabela contendo os tempos de execução correspondentes, permitindo o acompanhamento do progresso das otimizações realizadas.

Palavras-chave: Semigrupos numéricos. Otimização. Árvores. Conjectura de Bras-Amorós.

Abstract

This work refers to the development of connections between computational concepts and numerical semigroups, more specifically aiming to understand the counting of numerical semigroups by given genus. Here we will discuss the operations and techniques necessary to obtain elements of the sequence (n_g) quickly and with greater efficiency using programming languages. In this way, with the aid of previously elaborated studies, we seek to define the best computational alternatives both in terms of time and execution of operations and in relation to the particularity of programming languages. The construction of the tree of numerical semigroups was implemented in the programming languages Python and C++. Additionally, Cilk++ was employed for parallelizing the process. It is important to mention that for this research the concepts of tree of numerical semigroups were adopted as a mathematical reference. In addition to this, it relates to the methodologies created by Fromentin and Hivert in order to establish the computational and mathematical relationships objectified in this work. It should be noted that prior knowledge of both some algorithm structures and some mathematical concepts collaborate for the reading and understanding of this research. This research resulted in the computation of the sequence up to n_{64}, accompanied by the creation of a table containing the respective execution times, enabling the tracking of the progress made during the optimizations performed.

Key-words: Numerical semigroups. Optimization. Trees. Bras-Amorós Conjecture.

Lista de ilustrações

Figura 1 – Gráficos das funções do Exemplo 2.1 32
Figura 2 – Gráficos das funções do Exemplo 2.2 32
Figura 3 – Árvore de Semigrupos - Construção Ordinários 36
Figura 4 – Árvore de Semigrupos - Construção Ordinários 36
Figura 5 – Árvore de Semigrupos - Construção não-Ordinários 37
Figura 6 – Árvore de Semigrupos - Construção não-Ordinários 37
Figura 7 – Árvore de Semigrupos (até $g = 4$) 38
Figura 8 – Execução do Algoritmo - Parte 1 47
Figura 9 – Execução do Algoritmo - Parte 2 47
Figura 10 – Execução do Algoritmo - Parte 3 48
Figura 11 – Execução do Algoritmo - Parte 4 48
Figura 12 – Execução do Algoritmo - Final 49

Lista de tabelas

Tabela 1 – Tempos de Execução (em segundos) 59
Tabela 2 – Alguns valores de n_g 60

Lista de símbolos

$\mathbb{N}$	Conjunto dos números naturais - $\{1, 2, 3, \ldots\}$
$\mathbb{N}_0$	Conjunto dos números naturais incluindo zero - $\{0, 1, 2, 3, \ldots\}$
$\mathbb{Z}$	Conjunto dos números inteiros - $\mathbb{N}_0 \cup \{-n : n \in \mathbb{N}\}$
$[a, \infty)$	Conjunto dos números inteiros maiores que ou iguais a a
$\lceil x \rceil$	Menor inteiro maior que ou igual ao número real x
$\lfloor x \rfloor$	Maior inteiro menor que ou igual ao número real x
$\#X$	Cardinalidade de um conjunto X

Sumário

Introdução 19

1 FUNDAMENTAÇÃO TEÓRICA 21

1.1 Semigrupos numéricos 21

1.2 Invariantes 22

1.3 Paralelo entre os invariantes e os exemplos 23

1.4 Elementos irredutíveis de um semigrupo numérico 24

1.5 Proposições essenciais 25

2 PRELIMINARES COMPUTACIONAIS 29

2.1 Stack 29

2.2 Árvores 29

2.3 Busca em profundidade (DFS) 30

2.3.1 DFS e BFS: Decisão estratégica na travessia da árvore de semigrupos numéricos 30

2.4 Cilk++ 31

2.5 Análise de complexidade 31

2.5.1 Notação Big-O 32

3 ANÁLISE E IMPLEMENTAÇÃO DA ÁRVORE GERADORA DE SEMIGRUPOS NUMÉRICOS 35

3.1 Árvore dos semigrupos numéricos 35

3.2 Decomposição numérica 39

3.3 Algoritmos em python 41

3.3.1 Instância de um semigrupo numérico 42

3.3.2 Raiz da árvore 42

3.3.3 Ramificações 43

3.3.4 Construção da árvore 45

3.3.5 Análise de complexidade do algoritmo 49

3.4 Algoritmos em C++ 50

3.4.1 Instância de um semigrupo numérico 51

3.4.2 Raiz e ramificações da árvore 52

3.4.3 Exploração da árvore 53

3.5 Otimizações 54

3.5.1 SIMD, SSE e MMX 54

3.5.2 Vetorização 55

3.5.3 Paralelização 57
3.6 Tempos de execução 58
3.7 Execução do algoritmo 59

4 CONSIDERAÇÕES FINAIS 63
4.1 Trabalhos Futuros 63

REFERÊNCIAS 65

Introdução

Este trabalho apresenta considerações gerais e conceituações no que diz respeito aos semigrupos numéricos. Um semigrupo numérico é caracterizado por ser um subconjunto de $\mathbb{N}_0$ munido da adição usual de tal forma que sejam atendidas três propriedades, sendo elas: $0 \in S$; S é fechado para adição; o conjunto de lacunas, denotado por $\mathcal{L}(S) := \mathbb{N}_0 \setminus S$, é finito. A cardinalidade de $\mathcal{L}(S)$ é o gênero de S.

O estudo envolvendo tais conceitos iniciou-se no século XIX com o problema de Frobenius. Para obter uma visão aprofundada veja Sylvester (1884). Desde então, esta área de estudo se desenvolveu com novas descobertas em diversos tipos de problemas.

Além das propriedades citadas anteriormente, um semigrupo numérico possui alguns invariantes, que por sua vez serão utilizados para definições e compreensão de um semigrupo numérico. Portanto, no decorrer deste trabalho os conceitos de multiplicidade, condutor, gênero, profundidade e entre outros serão abordados e resgatados para entendimento do tema.

Um dos primeiros problemas que motivou o desenvolvimento do estudo dos semigrupos numéricos surge do problema de Frobenius, que pode ser ilustrado pela seguinte situação comum: realizar pagamento de um valor inteiro positivo, com notas de valores inteiros positivos sem utilizar troco. Em uma determinada situação deseja-se pagar o valor de 10 reais, sem receber troco, com notas de 3 e 7. Note que é possível, já que 10 = 3 + 7. Todavia não é possível pagar 8 reais com as mesmas notas sem receber troco. Desta forma, se questionado quais pagamentos não podem ser realizados com as notas de 3 e 7 sem receber troco, obtém-se a resposta: 1, 2, 4, 5, 8, 11. Nesse caso, temos uma quantidade finita. Contudo, se desejássemos usar apenas as notas de 2 e 4, por exemplo, nenhum valor ímpar poderia ser pago sem receber troco.

Este conceito permite criar teorias de fácil compreensão, mas com soluções não triviais. Vários matemáticos como Frobenius e Sylvester, no final do século XIX, se interessaram neste estudo. Assim surge o problema de Frobenius (Troco de Frobenius), o qual desejava encontrar uma fórmula dependendo de $n_1, \ldots, n_e$ para o maior inteiro não pertencente a $\langle n_1, \ldots, n_e \rangle$, conjunto de todas as combinações lineares não negativas de $n_1, \ldots, n_e$, em que $\text{mdc}(n_1, n_2, \ldots, n_e) = 1$.

O principal objeto de estudo desta monografia é a árvore de semigrupos numéricos, apresentada por Bras-Amorós. As características e relações da construção desta árvore serão exemplificadas em cada nível. Essa árvore é uma forma de organizar os semigrupos numéricos pelo gênero. Em 2008, Bras-Amorós utilizou dessa árvore para exibir os primeiros 50 elementos da sequência (n_g), em que n_g denota a quantidade de semigrupos

numéricos de gênero g, onde $g \in \mathbb{N}_0$. Ela conjecturou que:

1. $\lim_{g \to \infty} \frac{n_{g+1}}{n_g} = \frac{1+\sqrt{5}}{2}$;

2. $\lim_{g \to \infty} \frac{n_{g+1} + n_g}{n_{g+2}} = 1$;

3. $n_g + n_{g+1} \leq n_{g+2}, \forall g \in \mathbb{N}_0$.

A partir da exploração dessa árvore iremos compreender o trabalho de Fromentin e Hivert (2016) acerca da estratégia adotada na criação de um algoritmo eficiente de exploração da árvore. Importante ressaltar que as estruturas de dados e técnicas computacionais utilizadas para construção algorítmica, serão discutidas e expostas.

A seleção do tema de pesquisa tem como cerne relacionar conceitos matemáticos e computacionais, mais especificamente entender sobre a sequência da quantidade de semigrupos numéricos de gênero dado, em que g varia no conjunto dos inteiros não-negativos.

O objetivo geral da pesquisa é realizar a contagem da sequência (n_g) utilizando a árvore de semigrupos numéricos e o algoritmo proposto por Fromentin e Hivert (2016). Os objetivos específicos envolvem apresentar as ferramentas e técnicas utilizadas para otimização do algoritmo da árvore de semigrupos numéricos e também compreender a sua complexidade e desempenho.

Esta pesquisa é direcionada a acadêmicos, pesquisadores e estudantes nas áreas de Computação e Matemática, com interesse em semigrupos numéricos e estruturas de árvore. As implementações e resultados deste estudo têm por intuito contribuir com o conhecimento nessas áreas. Este estudo também pode ser relevante para pesquisadores que exploram tópicos relacionados, como algoritmos de construção de árvores e aplicações práticas de semigrupos numéricos.

Referente ao trabalho iremos relatar brevemente o que cada capítulo irá abordar.

- **Capítulo 1**: Nesta etapa iremos explicar as definições de um semigrupo numérico, seus invariantes e suas respectivas propriedades.

- **Capítulo 2**: Apresentaremos o conceito das técnicas e estruturas computacionais utilizadas nos algoritmos que serão apresentados.

- **Capítulo 3**: Apresentaremos árvore de semigrupos numéricos e os principais resultados para sua construção. Este capítulo também abordará o algoritmo apresentado no trabalho de Fromentin e Hivert.

1 Fundamentação Teórica

Neste capítulo iremos definir o que é um semigrupo numérico e seus respectivos invariantes. Também abordaremos o processo de construção da árvore de semigrupos numéricos e sua complexidade.

1.1 Semigrupos numéricos

Um semigrupo numérico S é um subconjunto de $\mathbb{N}_0$ munido da adição usual que deve satisfazer as seguintes propriedades:

- $0 \in S$;
- S é fechado para adição, ou seja, $x \in S$ e $y \in S$ implica que $x + y \in S$;
- O conjunto de lacunas, denotado por $\mathcal{L}(S) = \mathbb{N}_0 \backslash S$, é finito.

Os elementos de $\mathcal{L}(S)$ são chamados de lacunas de S e os elementos de S de não-lacunas. No decorrer deste trabalho, utilizaremos a seguinte notação: se $A = \{a_1, \ldots, a_n\} \subseteq \mathbb{Z}$ e $a \in \mathbb{Z}$ com $a_1 < a_2 < \ldots < a_n < a$, então denotaremos o conjunto $A \cup \{x \in \mathbb{Z} : x \geq a\}$ por $\{a_1, \ldots, a_n, a, \rightarrow\}$.

Exemplo 1.1. Considere o conjunto $S = \{0, 3, 5, \rightarrow\}$. Tem-se que

- $0 \in S$;
- S é fechado para adição: os números $0 + 0$, $0 + 3$, $3 + 3$ estão presentes no semigrupo e se $a, b \in S$ com $a \geq 5$ então $a + b \geq 5$ isto é, $a + b \in S$.
- $\mathcal{L}(S) = \{1, 2, 4\}$ é finito.

  Portanto, S é um semigrupo numérico.

Exemplo 1.2. Considere o semigrupo numérico $S = \{0, 2, \rightarrow\}$. Tem-se que

- $0 \in S$;
- S é fechado para adição: o número $0 + 0$, está presente no semigrupo e se $a, b \in S$ com $a \geq 2$ então $a + b \geq 2$ isto é, $a + b \in S$.;
- $\mathcal{L}(S) = \{1\}$ é finito.

  Portanto S é um semigrupo numérico.

1.2 Invariantes

Define-se também quais são as invariantes dos semigrupos numéricos e os conceitos relacionados a cada um. Seja S um semigrupo numérico. Utilizaremos os seguintes invariantes:

- O **gênero** de S é a cardinalidade de $\mathcal{L}(S)$, ou seja, é a quantidade de lacunas de S. O gênero de S é denotado por $g(S)$;
- A **multiplicidade** de S, denotada por $m(S)$, é o menor elemento não nulo de S;
- O **condutor** de S, $c(S)$, é o menor elemento que, a partir dele, todos estão em S;
- O **número de Frobenius** de S denotado por $F(S)$ é o maior elemento de $\mathbb{Z} \setminus S$; Caso $S \neq \mathbb{N}_0$, ele coincide com a maior lacuna de S. Note que $F(S) = c(S) - 1$. Vale ressaltar que $S = \mathbb{N}_0$, teremos que $F(\mathbb{N}_0) = -1$, sendo esse o único caso que o número de Frobenius é menor que zero.
- A **profundidade** de S, denotada por $q(S)$, é definida por $q(S) = \left\lceil \frac{c(S)}{m(S)} \right\rceil$.

Apresentamos os seguintes exemplos:

Exemplo 1.3. Considere o semigrupo numérico $S_a = \{0, 1, 2, 3, \rightarrow\}$. Nota-se que

- O conjunto de lacunas é $G(S_a) = \emptyset$;
- O gênero de S_a é $g(S_a) = 0$;
- A multiplicidade de S_a é $m(S_a) = 1$;
- O condutor de S_a é $c(S_a) = 0$ e, portanto, o número de Frobenius de S_a é $F(S_a) = -1$;

Exemplo 1.4. Considere o semigrupo numérico $S_b = \{0, 3, \rightarrow\}$. Nota-se que

- O conjunto de lacunas é $G(S_b) = \{1, 2\}$;
- O gênero de S_b é $g(S_b) = 2$;
- A multiplicidade de S_b é $m(S_b) = 3$;
- O condutor de S_b é $c(S_b) = 3$ e, portanto, o número de Frobenius de S_b é $F(S_b) = 2$;
- A profundidade de S_b é $q(S_b) = \lceil \frac{3}{3} \rceil = 1$.

Exemplo 1.5. Considere o semigrupo numérico $S_c = \{0, 2, 4, 6, 8, \rightarrow\}$. Nota-se que

- O conjunto de lacunas é $G(S_c) = \{1, 3, 5, 7\}$;
- O gênero de S_c é $g(S_c) = 4$;
- A multiplicidade de S_c é $m(S_c) = 2$;
- O condutor de S é $c(S_c) = 8$. Portanto, o número de Frobenius de S_c é $F(S_c) = 7$.
- A profundidade de S_c é $q(S_c) = \lceil \frac{8}{2} \rceil = 4$.

Exemplo 1.6. Considere o semigrupo numérico $S_d = \{0, 3, 6, 7, 9, 10, 12, \rightarrow\}$. Nota-se que

- O conjunto de lacunas é $G(S_d) = \{1, 2, 4, 5, 8, 11\}$;
- O gênero de S_d é $g(S_d) = 6$;
- A multiplicidade de S_d é $m(S_d) = 3$;
- O condutor de S_d é $c(S_d) = 12$. Portanto, o número de Frobenius de S é $F(S_d) = 11$.
- A profundidade de S_d é $q(S_d) = \lceil \frac{12}{3} \rceil = 4$.

Exemplo 1.7. Considere o semigrupo numérico $S_e = \{0, 6, 7, 12, \rightarrow\}$. Nota-se que:

- O conjunto de lacunas é $G(S_e) = \{1, 2, 3, 4, 5, 8, 9, 10, 11\}$;
- O gênero de S_e é $g(S_e) = 9$;
- A multiplicidade de S_e é $m(S_e) = 6$;
- O condutor de S_e é $c(S_e) = 12$. Portanto, o número de Frobenius de S é $F(S_e) = 11$.
- A profundidade de S_e é $q(S_e) = \lceil \frac{12}{6} \rceil = 2$.

1.3 Paralelo entre os invariantes e os exemplos

No Exemplo 1.3 em que $S_a = \mathbb{N}_0$, a multiplicidade assume valor 1. Em outros casos em que $S \neq \mathbb{N}_0$, o elemento 1 necessariamente tornar-se-ia uma lacuna, para mantermos a propriedade do fechamento para adição. Os invariantes permitem categorizar os semigrupos numéricos, e apresentam particularidades como cotas superiores e inferiores, que fornecem informações mais detalhadas sobre as características de um semigrupo numérico.

Conforme demonstrado em Júnior (2020), teremos o seguinte resultado.

Proposição 1.1. Considere um semigrupo numérico $S \neq \mathbb{N}_0$ de gênero g, multiplicidade m, condutor c e profundidade q. Então:

1. $2 \leq m \leq g + 1$;

2. $g + 1 \leq c \leq 2g$;

3. $1 \leq q \leq g$.

No Exemplo 1.4, o condutor de S_b atinge a cota inferior $g(S_b) + 1$. Nestes casos, temos uma definição especial para estes semigrupos. Seja S um semigrupo em que o conjunto de lacunas $\mathcal{L}(S)$ é $[1, g] \cap \mathbb{Z}$ e consequentemente $S = \{0\} \cup [g + 1, \infty) \cap \mathbb{Z}$. Este semigrupo numérico é chamado de *ordinário*. Vale ressaltar que, o Exemplo 1.4. também atinge a cota superior do primeiro item acerca da multiplicidade, onde $m(S_b) = g(S_b)+1 = 3$. Neste mesmo exemplo a profundidade $q(S_b) = 1$ e consequentemente atinge a cota inferior abordada no terceiro item. No Exemplo 1.5, S_c atinge a cota superior do condutor em que $2g(S_c) = 8$, sendo considerado um semigrupo *simétrico*. Um semigrupo é conhecido como *simétrico*, quando o condutor deste mesmo semigrupo atinge a cota superior $2g$. Também vale a pena ressaltar o conceito de semigrupos hiperelípticos, que consiste em semigrupos que não contêm os naturais ímpares menores que $2g$. S_c é um exemplo de semigrupo hiperelíptico e simétrico. É valido apontar que todo semigrupo hiperelíptico é simétrico, entretanto conforme observado no Exemplo 1.6 nem todo semigrupo simétrico é de fato hiperelíptico.

Ainda em S_c a multiplicidade atinge a cota inferior da primeira propriedade referente a multiplicidade tal que $m(S_c) = 2$. Também podemos destacar que a profundidade $q(S_c)$ atinge a cota superior do terceiro item, pois conforme observado, $q(S_c) = \left\lceil \frac{c(S_c)}{m(S_c)} \right\rceil = 4$.

O último item aborda a profundidade, a partir dos conceitos apresentados anteriormente, podemos notar que os semigrupos S_b e S_c representam respectivamente a cota inferior e superior da profundidade q. Tem-se, portanto, que para semigrupos caracterizados como *ordinários* temos que $q = 1$ e para os *hiperelípticos* temos $q = g$.

1.4 Elementos irredutíveis de um semigrupo numérico

Seja S um semigrupo numérico. Considera-se determinado elemento não nulo de S *irredutível* quando não é possível escrevê-lo como a soma de outros elementos não nulos pertencentes a S. Denotaremos o conjunto dos irredutíveis de S por $Irr(S)$.

A partir dos exemplos apresentados anteriormente é possível observar que:

- No Exemplo 1.3, tem-se que $Irr(S_a) = \{1\}$.
- No Exemplo 1.4, tem-se que $Irr(S_b) = \{3, 4, 5\}$.
- No Exemplo 1.5, tem-se que $Irr(S_c) = \{2, 9\}$.

- No Exemplo 1.6, tem-se que $Irr(S_d) = \{3, 7\}$.
- No Exemplo 1.7, tem-se que $Irr(S_e) = \{6, 7, 15, 16, 17\}$.

Seja S um semigrupo numérico. Dizemos que o conjunto $X = \{x_1, x_2, \ldots, x_n\} \subset S$ é um conjunto de geradores de S se todo elemento de S pode ser escrito como soma de elementos de X. Neste caso, escrevemos $S = \langle x_1, x_2, \ldots, x_n \rangle$. Em particular, é de interesse encontrar o conjunto minimal de geradores.

O conjunto minimal de geradores de um semigrupo numérico S é o menor (em relação à inclusão) conjunto de geradores; é uma forma de descrever um semigrupo S a partir de um conjunto finito. De acordo com Rodrigues (2020), pode-se dizer que todo elemento de S é uma combinação inteira não negativa dos elementos presentes no conjunto minimal de geradores.

Em Rosales e García-Sánchez (2009), mostra-se que se S é semigrupo numérico, então o conjunto dos elementos irredutíveis de S coincide com o conjunto minimal de geradores de S.

Exemplo 1.8. Podemos descrever os semigrupos numéricos dos Exemplos 1.3 ao 1.7 em função dos seus geradores minimais

- O semigrupo numérico S_a pode ser obtido a partir dos geradores minimais e escrevemos $S_a = \langle 1 \rangle$;
- O semigrupo numérico S_b pode ser obtido a partir dos geradores minimais e escrevemos $S_b = \langle 3, 4, 5 \rangle$;
- O semigrupo numérico S_c pode ser obtido a partir dos geradores minimais e escrevemos $S_c = \langle 2, 9 \rangle$;
- O semigrupo numérico S_d pode ser obtido a partir dos geradores minimais e escrevemos $S_d = \langle 3, 7 \rangle$;
- O semigrupo numérico S_e pode ser obtido a partir dos geradores minimais e escrevemos $S_e = \langle 6, 7, 15, 16, 17 \rangle$.

1.5 Proposições essenciais

A operação de retirar um elemento de um semigrupo numérico desempenha um papel fundamental na criação de um novo semigrupo numérico. Ao remover o elemento de um semigrupo numérico com alguma restrição, é possível construir um novo conjunto de elementos que mantém as propriedades. Ao longo deste trabalho utilizaremos a seguinte

notação para representar a criação de um novo conjunto a partir de um semigrupo numérico S pela retirada de um elemento $x \in S$: $S^x = S \backslash \{x\}$. Nessa representação, o símbolo S^x representa o novo conjunto obtido a partir do semigrupo numérico S ao remover o elemento x.

Exemplo 1.9. Considere o semigrupo numérico $S_b = \{0, 3, \rightarrow\}$. Seja x o elemento a ser retirado de S_b.

- $x = 3$:

  Verifica-se que $S_b^3 = \{0, 4, \rightarrow\}$, que por usa vez, atende as propriedades de um semigrupo numérico. Portanto, S_b^3 é semigrupo numérico.

- $x = 4$:

  Verifica-se que $S_b^4 = \{0, 3, 5, \rightarrow\}$, que mantém todas as propriedades de um semigrupo numérico. Logo, podemos concluir que S_b^4 é um semigrupo numérico.

- $x = 5$:

  Tem-se que $S_b^5 = \{0, 3, 4, 6, \rightarrow\}$, que também mantém todas as propriedades de um semigrupo numérico. Sendo assim, S_b^5 é um semigrupo numérico.

- $x = 6$:

  Tem-se que $S_b^6 = \{0, 3, 4, 5, 7, \rightarrow\}$. Entretanto, S_b^6 não é um semigrupo numérico, pois $3 + 3 = 6 \notin S_b^6$ não respeitando a propriedade de fechamento para adição.

A partir dos exemplos e conceitos estabelecidos no decorrer deste capítulo, a seguinte proposição de Hivert e Fromentin conseguem abordar as principais características dos invariantes que utilizaremos no desenvolvimento deste trabalho.

Proposição 1.2. (Proposição 1.5 de Fromentin e Hivert (2016)) Seja S um semigrupo numérico. Então $S^x = S \backslash \{x\}$ é um semigrupo numérico se, e somente se, $x \in Irr(S)$.

Demonstração: ($\Leftarrow$) Considere as seguintes hipóteses:

1. $0 \in S$;
2. $a, b \in S \Rightarrow a + b \in S$;
3. $\mathbb{N}_0 \setminus S$ é finito;
4. $x \in Irr(S)$.

A tese aborda as seguintes propriedades

a. $0 \in S^x$.

b. $a, b \in S^x \Rightarrow a + b \in S^x$.

c. $\mathbb{N}_0 \setminus S^x$ é finito.

Podemos elucidar a partir de cada item da hipótese que

a. Como $0 \in S$ e $x \neq 0$, então $0 \in S^x$.

b. Sejam $a, b \in S^x = S \setminus \{x\}$. Logo, $a, b \in S$. Pela hipótese, $a + b \in S$. Basta provar que $a + b \in S \setminus \{x\}$. Observe que $a + b = x$ é impossível, pois x é um elemento irredutível de S. Logo $a + b \in S^x$.

c. Note que, se $\#\mathbb{N}_0 \setminus S = g$, então $\#\mathbb{N}_0 \setminus S^x = g + 1$ e $\mathbb{N}_0 \setminus S^x$ permanece um conjunto finito.

($\Rightarrow$) Considere as seguintes hipóteses:

1. $0 \in S^x$;

2. $a, b \in S^x \Rightarrow a + b \in S^x$;

3. $\mathbb{N}_0 \setminus S^x$ é finito;

Queremos mostrar que $x \in Irr(S)$. Suponha que $x \notin Irr(S)$. Então, $x = y + z$, com $y, z \in S \setminus \{0\}$. Observe que, $y, z \neq x$ e, por isso, $y, z \in S^x = S \setminus \{x\}$. Como S^x é um semigrupo numérico, é possível afirmar que $y + z \in S^x$, isto é, $y + z \neq x$, o que é uma contradição. Portanto, $x \in Irr(S)$.

□

Proposição 1.3. (Proposição 1.4 de Fromentin e Hivert (2016)) Sejam S um semigrupo numérico e $x \in Irr(S)$. Então $x \leq c(S) + m(S) - 1$.

Demonstração: Vamos mostrar pela contra-recíproca. Seja $x > c(S) + m(S) - 1$. Então $x = c(S) + m(S) - 1 + y$, em que $y \in \mathbb{N}$. Podemos reescrever $x = m(S) + [c(S) + y - 1]$. Note que $y - 1 \geq 0$, $m(S) \in S$ e $[c(S) + y - 1] \in S$. Logo, como x pode ser obtido a partir da soma dos elementos em S, concluímos que x não é irredutível.

□

2 Preliminares Computacionais

Nessa seção tem-se por objetivo explicar e esclarecer os conceitos gerais sobre as técnicas e estruturas computacionais que estão associadas ao desenvolvimento deste trabalho. As definições citadas podem ser mais aprofundadas em Halim e Halim (2013) e Cormen et al. (2009).

2.1 Stack

Uma pilha (ou *stack* em inglês) é uma estrutura de dados abstrata (ADT - *Abstract Data Type*) que segue o princípio **LIFO** (*Last In, First Out*), o que significa que o último elemento inserido é o primeiro a ser removido. A pilha possui diversas operações, sendo essas as principais operações necessárias para a construção da árvore de semigrupos numéricos:

- *Push*: insere um elemento no topo da pilha;
- *Pop*: remove o elemento mais recente da pilha;
- *Empty*: Verifica se a pilha está vazia ou não;
- *Top*: Retorna o elemento que está no topo da pilha;
- *Size*: A quantidade de elementos armazenados na pilha.

Todas as operações listadas anteriormente possuem uma complexidade de tempo constante $O(1)$. A pilha desempenha um papel fundamental no algoritmo, pois permite armazenar os semigrupos numéricos durante a execução do algoritmo, o que possibilita a contagem dos elementos na sequência (n_g).

2.2 Árvores

As árvores são estruturas compostas por nós e arestas, e sua visualização em algoritmos é invertida, com a raiz no topo e as folhas na base. A raiz é um nó que não tem pai, enquanto as folhas são nós que não possuem filhos. Os nós que contêm filhos são chamados nós não-terminais ou nós internos.

Cada nó pode ser alcançado, a partir da raiz, por meio de uma única sequência de ramos, denominada caminho. O nível de um nó N corresponde ao número de nós no caminho de N até a raiz. A altura de uma árvore é igual ao nível máximo dentre todos os

nós da árvore, isto é, a altura de uma árvore corresponde à distância entre sua raiz e sua folha mais distante, representando a profundidade máxima da árvore. Uma árvore sem nós é considerada de altura zero, enquanto uma árvore com apenas um nó possui altura 1.

A compreensão do conceito de árvore é fundamental, uma vez que neste trabalho iremos explorar a árvore de semigrupos numéricos, que incorpora todas as características mencionadas anteriormente, e organiza o conjunto de semigrupos numéricos pelo gênero.

2.3 Busca em profundidade (DFS)

DFS (*Depth-First Search*), ou Busca em Profundidade, é um algoritmo de busca utilizado para percorrer estruturas de dados como árvores e grafos. Ele explora os elementos da estrutura seguindo um caminho o mais profundo possível antes de retroceder e explorar outros caminhos. Em árvores, o DFS começa a partir da raiz e percorre cada ramificação o mais profundamente possível, até encontrar o alvo da busca ou até chegar a um nó que não possui mais ramificações. Nesse ponto, o algoritmo retrocede (*backtracking*) e continua explorando outros caminhos ainda não percorridos.

A DFS é amplamente utilizado em problemas de busca, como encontrar um caminho entre dois nós. Ele é implementado de maneira recursiva ou por meio de uma pilha. Em resumo, a busca em profundidade é uma estratégia que segue uma abordagem de explorar o máximo possível em um caminho antes de retroceder.

No âmbito do nosso contexto, a técnica da busca em profundidade será empregada na exploração da árvore de semigrupos numéricos.

2.3.1 DFS e BFS: Decisão estratégica na travessia da árvore de semigrupos numéricos

A Busca em Largura (BFS) é um algoritmo eficiente para explorar grafos e árvores, garantindo que os nós mais próximos do nó inicial sejam visitados primeiro. No entanto, sua característica de armazenar todos os nós de um determinado nível antes de avançar para o próximo nível pode levar a um alto consumo de memória, especialmente em árvores extensas. Diante dessa limitação, torna-se viável utilizar a Busca em Profundidade (DFS) na árvore de semigrupos numéricos tendo em vista que a DFS explora a estrutura de forma mais compacta, percorrendo um ramo até sua profundidade máxima permitindo assim uma economia de memória ao não armazenar todos os nós de um nível.

Além das considerações mencionadas anteriormente sobre a eficiência de memória, a paralelização da Busca em Profundidade (DFS) também apresenta algumas vantagens em relação à paralelização da Busca em Largura (BFS). Ao paralelizar a DFS, cada

ramo explorado é tratado de forma independente, facilitando a divisão das tarefas entre os diferentes núcleos de processamento. Além disso, a DFS geralmente apresenta menos sobrecarga associada à comunicação e coordenação entre os processos, contribuindo para uma paralelização mais eficiente. Por outro lado, na paralelização da BFS, a natureza sequencial de explorar os nós por níveis pode introduzir limitações na paralelização, já que os níveis precisam ser processados em ordem e a sincronização entre os processos torna-se mais complexa.

A escolha da DFS em detrimento da BFS, mostrou-se mais adequada para o contexto do algoritmo de construção da árvore de semigrupos numéricos, proporcionando uma execução mais eficiente e satisfatória em relação ao uso de recursos computacionais.

2.4 Cilk++

Cilk é uma linguagem de programação baseada em C desenvolvida para computação paralela *multithread* . A criação começou em meados dos anos 90 como um projeto no laboratório do MIT. Inicialmente a tecnologia Cilk forneceu redução do tempo de execução comprovadamente eficiente, porém pouco suporte linguístico. Atualizações seguintes forneceram as extensões linguísticas necessárias, como a extensão do modelo Cilk para C++.

Posteriormente o MIT licenciou a tecnologia *Cilk* para a *Cilk Arts*, que desenvolveu o *Cilk ++*, uma melhoria que inclui suporte para C++, loops paralelos e interoperabilidade superior com código serial.

Ao usar a linguagem *Cilk* o programador deve ser responsável por expor o paralelismo, identificando elementos que possam ser executados em paralelo. Então o escalonador decide como dividir o trabalho entre os processadores.

2.5 Análise de complexidade

A análise de complexidade é uma técnica utilizada para medir o desempenho de um algoritmo, especialmente em relação ao tempo de execução e utilização dos recursos computacionais. O objetivo da análise de complexidade é identificar como os algoritmos se comportam em relação ao tamanho da entrada, seu custo computacional e qual seria o comportamento desta função à medida que o tamanho da entrada tende ao infinito.

Neste trabalho, utilizaremos a complexidade assintótica para descrever o tempo de execução de um algoritmo e também o espaço em memória que o algoritmo utilizará durante sua execução.

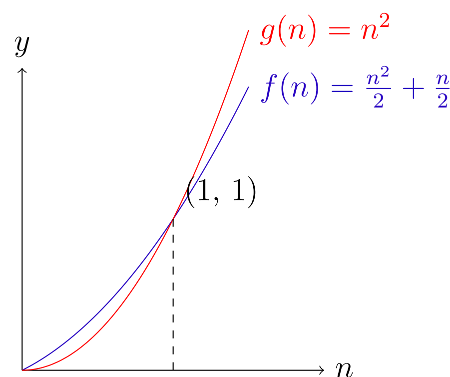
2.5.1 Notação Big-O

Considerando o apresentado no trabalho de Cormen et al. (2009), a notação big-O é uma ferramenta utilizada para descrever o comportamento assintótico de uma função, concentrando-se no seu limite superior. Em outras palavras, ela oferece uma estimativa de como a função cresce à medida que seus argumentos aumentam para valores muito grandes.

Dizemos que $f(x) = O(g(x))$ se existem $K \in \mathbb{R}$, $K > 0$, $x_0 \in \mathbb{R}$ tal que $|f(x)| \leq Kg(x)$, para todo $x \geq x_0$.

Exemplo 2.1. Considere a função dada por $f(n) = \frac{n^2}{2} + \frac{n}{2}$. Tem-se que $f(n)$ é $O(n^2)$, pois $f(n) \leq n^2$ para $n \geq 1$, conforme visto na Figura 1.

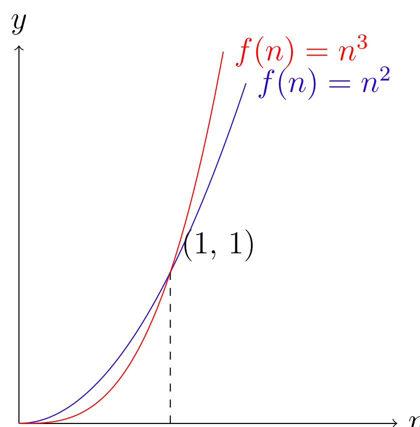
Figura 1 – Gráficos das funções do Exemplo 2.1



Fonte: O autor

Exemplo 2.2. Considere $f(n) = n^2$. Podemos afirmar que $f(n) = O(n^3)$ conforme visto na Figura 2.

Figura 2 – Gráficos das funções do Exemplo 2.2



Fonte: O autor

Mediante a notação big-O, podemos descrever o tempo de execução de um algoritmo analisando sua estrutura. Por exemplo, considere uma estrutura de laço duplo aninhada que escreverá todos os pares naturais (a, b) tais que $a < b \leq N$, para determinado valor de N.

Código 1 – Código C++ para exemplo de análise de complexidade.

```
1 vector<pair<int, int>> pares;
2
3 for (int a = 1; a <= N; a++)
4     for(int b = a + 1; b <= N; b++)
5         pares.push_bash(make_pair(a,b));
```

Fonte: O autor

O Código 1 contém dois laços alinhados. O primeiro laço é executado N vezes, e o segundo laço é executado $N - a$ vezes. No laço externo (linha 3) realizam-se duas operações, o incremento de a e a inicialização de b. No laço interno (linha 4) teremos o incremento de b e a inserção dos valores de a e b no vetor de pares.

O número total de iterações executadas é dada por

$$\sum_{i=1}^{N-1} i = 1 + 2 + 3 + \ldots + (N-1) = \frac{N(N-1)}{2}$$

Portanto, a complexidade assintótica do algoritmo é $O(N^2)$, uma vez que o número total de iterações é igual a $\frac{N(N-1)}{2}$, que cresce quadraticamente com o tamanho da entrada N.

Através da notação big-O, é possível observar um limite superior $O(N^2)$ no tempo de execução do pior caso, considerando que o custo de cada iteração e atribuições dos laços são constantes em $O(1)$. Além disso, é importante destacar que os índices a e b possuem um limite máximo de N e o tempo de execução do algoritmo varia para diferentes entradas de tamanho N.

3 Análise e Implementação da Árvore Geradora de Semigrupos Numéricos

Neste capítulo enfatizaremos o objetivo central do trabalho com auxílio dos conceitos apresentados anteriormente. Apresentaremos também algumas proposições que auxiliam na ideia central do algoritmo e sua implementação. Este capítulo é baseado no trabalho de Fromentin e Hivert (2016).

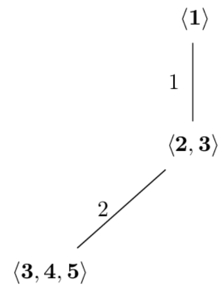
3.1 Árvore dos semigrupos numéricos

O principal objeto de estudo deste trabalho refere-se a árvore de semigrupos numéricos. A árvore é caracterizada por apresentar os semigrupos numéricos pelos geradores em cada nível representado pelo gênero g. Para construção da árvore adotamos o elemento base, ou raiz da árvore, como $\langle \mathbf{1} \rangle$ e $g = 0$ pois é a forma de representar o conjunto $\mathbb{N}_0$. No próximo nível aparece o semigrupo numérico $\langle \mathbf{2}, \mathbf{3} \rangle$ com $g = 1$ que representa o semigrupo numérico $\mathbb{N}_0 \backslash \{1\}$.

Os semigrupos numéricos podem ser categorizados como semigrupos *ordinários* e *não ordinários*. A ramificação mais à esquerda, que percorre os semigrupos numéricos $\langle \mathbf{2}, \mathbf{3} \rangle$, $\langle \mathbf{3}, \mathbf{4}, \mathbf{5} \rangle$, $\langle \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7} \rangle$, $\langle \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{9} \rangle$ e assim por diante contêm os semigrupos numéricos *ordinários*. A construção dos geradores dos semigrupos numéricos ordinários segue o seguinte raciocínio: retiramos o menor elemento irredutível x do semigrupo numérico ordinário anterior, e o próximo semigrupo numérico ordinário será $\langle x+1, \ldots, 2x+1 \rangle$. Também é possível obter o semigrupo numérico ordinário no nível g como $\langle g+1, \ldots, 2g+1 \rangle$. Esta demonstração pode ser encontrada em Júnior (2020).

Para exemplificar a construção dos semigrupos ordinários podemos utilizar $\langle \mathbf{2}, \mathbf{3} \rangle$. Este semigrupo numérico é ordinário e $g = 1$. Conforme citado anteriormente, podemos obter o semigrupo numérico ordinário quando $g = 2$ utilizando $\langle g+1, \ldots, 2g+1 \rangle = \langle \mathbf{3}, \mathbf{4}, \mathbf{5} \rangle$.

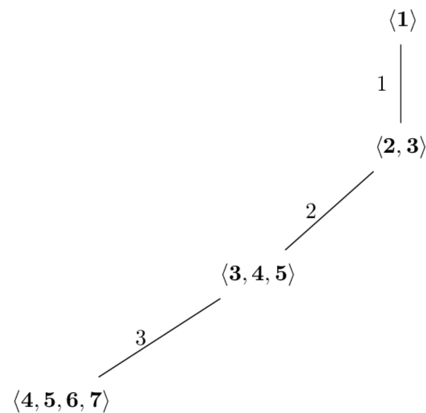
Figura 3 – Árvore de Semigrupos - Construção Ordinários



Fonte: Fromentin e Hivert (2016)

De outro modo apresentado, podemos gerar o semigrupo numérico ordinário quando $g = 3$ da seguinte forma: utilizamos como referência o semigrupo numérico $\langle \mathbf{3}, \mathbf{4}, \mathbf{5} \rangle$ anteriormente exemplificado, e a partir dele retiramos o elemento irredutível $x = 3$, mantendo assim os elementos $\langle \mathbf{4}, \mathbf{5} \rangle$, após isso inserimos os elementos que faltam até $2x + 1 = 7$, logo teremos o novo semigrupo numérico na seguinte forma $\langle \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7} \rangle$. Desta forma conseguimos gerar a ramificação referente aos semigrupos ordinários.

Figura 4 – Árvore de Semigrupos - Construção Ordinários

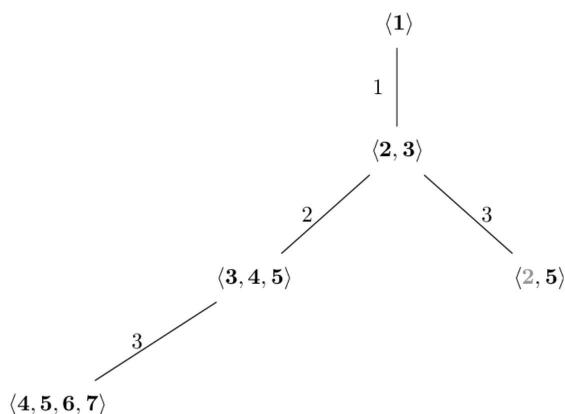


Fonte: Fromentin e Hivert (2016)

Para a construção dos semigrupos numéricos *não ordinários*, iremos adotar o seguinte algoritmo para produzir os próximos níveis: cada elemento irredutível do semigrupo numérico poderá criar uma nova ramificação. Previamente em $\langle \mathbf{2}, \mathbf{3} \rangle$, vimos que a retirada do elemento **2**, formou o semigrupo numérico ordinário $\langle \mathbf{3}, \mathbf{4}, \mathbf{5} \rangle$, agora no próximo exemplo, a retirada do elemento **3** irá gerar um novo semigrupo numérico não-ordinário. Vale ressaltar que o elemento a ser retirado em cada etapa da construção da árvore, é o número de Frobenius do semigrupo a ser inserido na árvore. Dando continuidade ao exemplo, considere o semigrupo numérico $\langle \mathbf{2}, \mathbf{3} \rangle$ e 3 como elemento irredutível a ser retirado. Para isso, seguiremos o seguinte raciocínio: gerar uma cópia do semigrupo numérico de referência sem o elemento 3. Por conseguinte, teremos que o novo semigrupo numérico será temporariamente composto apenas por $\langle 2 \rangle$. A última etapa a ser realizada, consiste

em adicionarmos ao novo gerador o elemento resultante da soma entre o elemento retirado e seu menor elemento irredutível. Portanto, tendo como base o elemento 3 e o elemento 2, teremos o novo semigrupo numérico $\langle 2, \mathbf{5} \rangle$.

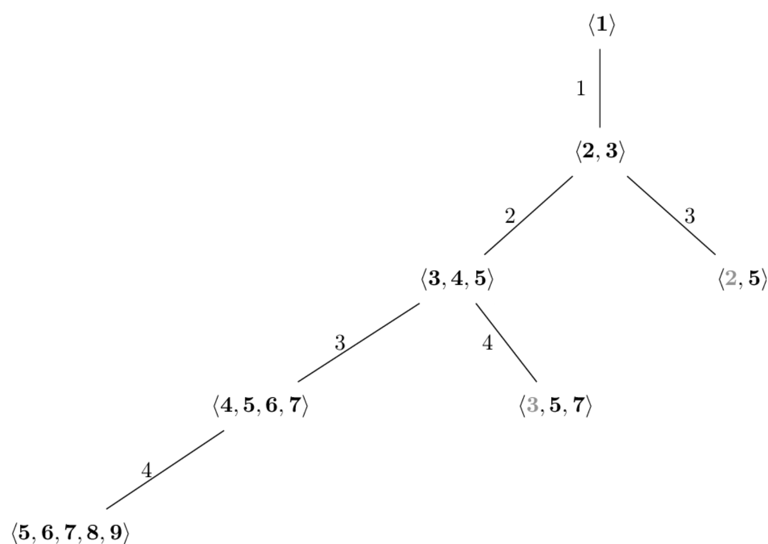
Figura 5 – Árvore de Semigrupos - Construção não-Ordinários



Fonte: Fromentin e Hivert (2016)

Outro exemplo seria realizar esse processo com o semigrupo numérico $\langle \mathbf{3}, \mathbf{4}, \mathbf{5} \rangle$, mas agora utilizaremos o elemento **4** como referência e consequentemente será o número de Frobenius do novo semigrupo numérico. O primeiro passo é gerar uma cópia desse semigrupo numérico sem a presença do elemento **4**, com isso teremos que o novo semigrupo numérico será temporariamente composto por $\langle 3, \mathbf{5} \rangle$. Conforme o apresentado anteriormente, iremos adicionar ao novo semigrupo a soma entre o menor elemento irredutível, neste caso 3, e o elemento retirado 4. Teremos, assim, que o novo semigrupo numérico é $\langle 3, \mathbf{5}, \mathbf{7} \rangle$ conforme pode ser observado na Figura 6.

Figura 6 – Árvore de Semigrupos - Construção não-Ordinários



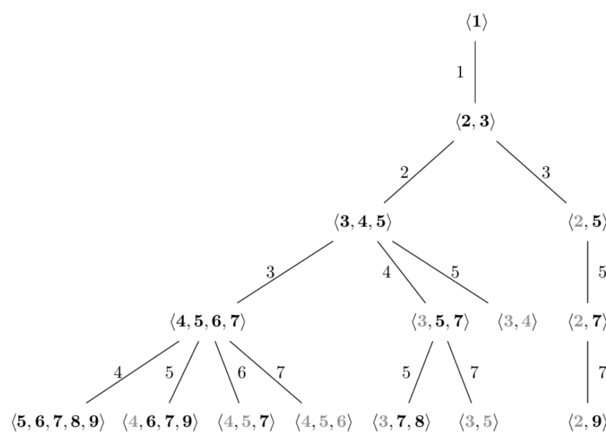
Fonte: Fromentin e Hivert (2016)

Aplicando este mesmo processo no elemento **5**. Gera-se uma cópia desse semigrupo numérico sem a presença do elemento **5**, assim, provisoriamente teremos $\langle 3, 4\rangle$. O próximo passo será adicionar ao novo semigrupo numérico, a soma entre o menor elemento irredutível, neste caso 3, e o elemento retirado 5. Teremos assim, que o novo semigrupo numérico é $\langle 3, 4, 8\rangle$. Como $8 = 4 + 4$ não é irredutível, concluímos, portanto, que o novo semigrupo numérico é $\langle 3, 4\rangle$.

Observação 3.1. Os elementos capazes de gerar outro semigrupo numérico, são os elementos que, necessariamente, são maiores que o número de Frobenius utilizado, para evitar repetição nas demais ramificações da árvore. Portanto, no exemplo acima, ao utilizarmos o elemento 2 como base, os elementos resultantes são aptos para gerarem novos semigrupos, no caso os elementos irredutíveis de $\langle \mathbf{3}, \mathbf{4}, \mathbf{5}\rangle$ são maiores que 2. Por outro lado, quando utilizamos o elemento 3 como base, vemos que o gerador de semigrupo resultante $\langle 2, \mathbf{5}\rangle$, apenas o elemento 5 é maior que o número de Frobenius utilizado para construção do ramo. Portanto, em um próximo nível apenas o elemento 5 irá ser capaz de gerar outro semigrupo para a árvore. Conforme visto na Proposição 1.2 os elementos capazes de gerar outros semigrupos, são os elementos irredutíveis. Para evitar repetições, devem ser maiores que ou iguais ao condutor.

Por fim, ao aplicarmos o algoritmo podemos construir a árvore de semigrupos numéricos em suas ramificações ordinárias e não ordinárias conforme a Figura 7.

Figura 7 – Árvore de Semigrupos (até $g = 4$)



Fonte: Fromentin e Hivert (2016)

3.2 Decomposição numérica

A decomposição numérica nos diz como um determinado valor $x \in \mathbb{N}_0$ pode ser descrito ou decomposto como a soma de dois elementos de S. Por definição, se $x \notin S$, então x não pode ser decomposto como soma de dois elementos de S. Fromentin e Hivert atribuíram em seu trabalho, a seguinte definição:

$$D_S(x) = \{y \in S \mid x - y \in S \text{ e } 2y \leq x\},$$

em que S é um semigrupo numérico e $x \in \mathbb{N}_0$. Vale observar que, se $x \notin S$ têm se que, $D_S(x) = \emptyset$. Essa definição é a base para construção do algoritmo da árvore. Um complemento desta definição é $d_s(x)$ que denota a cardinalidade de $D_S(x)$. Note que $D_S(x)$ é sempre finito e todos os elementos são menores que ou iguais a $\lfloor \frac{x}{2} \rfloor$.

Exemplo 3.1. Para melhor compreensão, considere $S_e = \{0, 6, 7, 12, 13, \rightarrow\}$ *e* $x = 13$; Basta verificar, dentre os números y de 0 a $\lfloor \frac{13}{2} \rfloor = 6$, quais deles estão em S e satisfazem a propriedade $13 - y \in S_e$.

- Para $y = 0$, tem-se que $2y = 0 < 13$. Como $13 - 0 = 13 \in S_e$, então $0 \in D_{S_e}(13)$.
- Para $y = 6$, tem-se que $2y = 12 < 13$. Como $13 - 6 = 7 \in S_e$, então $6 \in D_{S_e}(13)$.
- Podemos escolher $y = 7$, logo $2y = 14$. Entretanto, a restrição $2y \leq x$ inviabiliza esta escolha e isso vale para todo $y \geq 7$.

  A partir do exemplo podemos observar que para decompor x temos as opções $13 + 0 = 13$ *e* $13 = 6 + 7$. Portanto, $D_{S_e}(13) = \{0, 6\}$.

  A quantidade de maneiras a se obter x, a partir da decomposição numérica, nos ajuda a encontrar o valor de $d_{S_e}(x)$. Portanto, para $D_{S_e}(13) = \{0, 6\}$, tem-se que $d_{S_e}(13) = 2$.

Exemplo 3.2. Considere $S_e = \{0, 6, 7, 12, 13, \rightarrow\}$ *e* $x = 7$.

- Para $y = 0$, tem-se que $2y = 0$. Como $7 - 0 = 7 \in S_e$, então $0 \in D_{S_e}(7)$
- É possível notar que para qualquer outro elemento $y \neq 0$ teríamos o não cumprimento da restrição $2y \leq x$, portanto $D_{S_e}(7) = \{0\}$.

Ainda acerca da cardinalidade $d_S(x)$, conforme provado no trabalho de Fromentin e Hivert (2016), tem-se que $d_S(x) \leq 1 + \left\lfloor \frac{x}{2} \right\rfloor$.

A decomposição numérica possui duas consequências,

Proposição 3.1. Seja S um semigrupo numérico e $x \in \mathbb{N}$. Então

1. $x \in S$ se, e somente se, $d_S(x) > 0$;
2. $d_S(x) = 1$ se, e somente se, $x \in Irr(S)$.

Observação 3.2. É possível notar que $d_S(0) = 1$ afinal a única maneira de decompor o 0 em soma de dois elementos é $0 = 0 + 0$. Desconsideraremos o zero como um elemento minimal.

Proposição 3.2. (Proposição 2.5 Fromentin e Hivert (2016)) Considere um semigrupo numérico S e x um elemento irredutível de S. Então para todo $y \in \mathbb{N}$ temos

$$d_{S^x}(y) = \begin{cases} d_S(y) - 1 & se\ y \geq x\ e\ d_S(y-x) > 0, \\ d_S(y) & caso\ contrário. \end{cases}$$

O principal objetivo da decomposição numérica em dois elementos é justamente utilizar um conjunto finito de números para caracterizar todo o semigrupo numérico. A próxima proposição é o núcleo para a ideia expressa nos algoritmos utilizados.

Proposição 3.3. (Proposição 3.1 Fromentin e Hivert (2016)) Seja G um número inteiro e S um semigrupo numérico de gênero g, com $0 \leq g \leq G$. Então é possível descrever S em função do vetor de decomposição numérica $\delta_S = (d_S(0), \ldots, d_S(3G))$. Mais ainda, a partir de δ_S é possível obter $c(S),\ g(S),\ m(S)\ e\ Irr(S)$.

Observação 3.3. Conforme visto anteriormente, a partir dos elementos irredutíveis podemos descrever todo o semigrupo. A cota superior referente aos invariantes e aos elementos irredutíveis na Proposição 1.2 nos diz que se $x \in Irr(S)$, então $x \leq c(S)+m(S)-1$; Logo, substituindo cada invariante e suas respectivas cotas superiores conforme a Proposição 1.1, tem-se que $x \leq 2g(S) + g(S) + 1 - 1 = 3g(S)$. Logo, $x \leq 3G$. Por isso é necessário calcular $d_S(0)$ até $d_S(3G)$, para construção do vetor δ_S de forma a garantir que todos os irredutíveis sejam encontrados.

Para obtermos os invariantes de um semigrupo, a partir dos elementos de δ_S, têm-se as seguintes propriedades:

1. Para encontrar a multiplicidade é necessário resgatar o índice do primeiro elemento diferente de zero. Assim,

$$m(S) = \min\{i \in \{0, \ldots, 3G\},\ d_S(i)\ > 0\}.$$

2. Para encontrar o condutor, basta somar 1 ao maior índice cujo valor é zero. Assim

$$c(S) = 1 + \max\{i \in \{0, \ldots, 3G\},\ d_S(i)\ = 0\}.$$

3. Para obter o gênero, basta observar a quantidade de índices cujo valor é zero. Assim,

$$g(S) = \#\{i \in \{0, \dots, 3G\},\ d_S(i)\ = 0\}.$$

4. Para obter os elementos irredutíveis de S, conforme visto na Proposição 3.1, determinamos:

$$Irr(S) = \{i \in \{0, \dots, 3G\}, d_S(i)\ = 1\}.$$

Agora que é possível compreender a construção de um semigrupo numérico nos termos de δ_S, torna-se viável demonstrar a aplicação dos conceitos apresentados de maneira computacional.

Exemplo 3.3. Considere o semigrupo numérico $S_b = \{0, 3, \rightarrow\}$ cujo gênero é 2 e tome $G = 2$. Vamos obter δ_{S_b};

- $D_{S_b}(0) = \{0\}$ *e* $d_{S_b}(0) = 1$
- $D_{S_b}(1) = \emptyset$ *e* $d_{S_b}(1) = 0$
- $D_{S_b}(2) = \emptyset$ *e* $d_{S_b}(2) = 0$
- $D_{S_b}(3) = \{0\}$ *e* $d_{S_b}(3) = 1$
- $D_{S_b}(4) = \{0\}$ *e* $d_{S_b}(4) = 1$
- $D_{S_b}(5) = \{0\}$ *e* $d_{S_b}(5) = 1$
- $D_{S_b}(6) = \{0, 3\}$ *e* $d_{S_b}(6) = 2$

Portanto, $\delta_{S_b} = (1,\ 0,\ 0,\ 1,\ 1,\ 1,\ 2)$.

3.3 Algoritmos em python

Neste tópico serão apresentados os códigos gerados para realização da contagem, e referenciar as proposições levantadas anteriormente no algoritmo e suas respectivas funções.

Inicialmente, os algoritmos foram desenvolvidos utilizando Python como linguagem de programação devido a uma combinação de fatores, incluindo a facilidade de assimilação da sintaxe e a grande quantidade de bibliotecas e módulos disponíveis para realizar tarefas específicas, o que tornou o processo de desenvolvimento mais eficiente e produtivo.

Como visto no decorrer deste trabalho a construção de um semigrupo pode ser obtida mediante o conhecimento de seus elementos irredutíveis, a partir das informações de seus invariantes e mais recentemente abordado seu vetor δ_S.

3.3.1 Instância de um semigrupo numérico

Para representarmos um semigrupo numérico do ponto de vista computacional, o consideramos como um objeto e neste objeto temos justamente seus atributos, que ajudam a caracterizá-lo. Portanto, o código demonstra como é construído um semigrupo numérico e como é instanciado.

Código 2 – Código Python para instanciar semigrupos numéricos.

```
1 class NumericalSemigroup:
2
3     def __init__(self, condutor, genero, multiplicidade, G = 0):
4         self._c = condutor
5         self._g = genero
6         self._m = multiplicidade
7         self._vetor_decomposicao = []  # vetor de cardinalidades
8
9         self._Root(G)
```

Fonte: O autor

Este trecho de código refere-se ao método construtor que nos permite instanciar cada semigrupo e seus respectivos invariantes, agora é possível criar os semigrupos numéricos. É válido gerar o primeiro objeto S da árvore quando $g = 0$, isto é, instanciar a raiz da árvore quando temos $S = \mathbb{N}_0$. A complexidade desse algoritmo é constante, ou seja, $O(1)$ pois realiza apenas atribuições dos valores referentes aos invariantes.

3.3.2 Raiz da árvore

Código 3 – Código Python para atribuir o vetor de decomposição da raiz da árvore.

```
1 def _Root(self, G):
2     for x in range(0, 3 * G + 1):
3         self.vetor_decomposicao.append(1 + x // 2)
```

Fonte: O autor

O Código 3 é um método privado da classe apresentada no Código 2, responsável por atribuir os valores das cardinalidades ao vetor de decomposição numérica da raiz da árvore. O primeiro passo para construção da árvore é criar e armazenar os semigrupos numéricos e a partir deles gerar outros semigrupos numéricos, seguindo a mesma linha de raciocínio para construção da árvore a partir do G solicitado.

Exemplo 3.4. Considere $G = 3$. E vamos obter $\delta_{\mathbb{N}_0}$;

- $D_S(0) = \{0\}$ *e* $d_S(0) = 1$
- $D_S(1) = \{0\}$ *e* $d_S(1) = 1$
- $D_S(2) = \{0, 1\}$ *e* $d_S(2) = 2$
- $D_S(3) = \{0, 1\}$ *e* $d_S(3) = 2$
- $D_S(4) = \{0, 1, 2\}$ *e* $d_S(4) = 3$
- $D_S(5) = \{0, 1, 2\}$ *e* $d_S(5) = 3$
- $D_S(6) = \{0, 1, 2, 3\}$ *e* $d_S(6) = 4$
- $D_S(7) = \{0, 1, 2, 3\}$ *e* $d_S(7) = 4$
- $D_S(8) = \{0, 1, 2, 3, 4\}$ *e* $d_S(8) = 5$
- $D_S(9) = \{0, 1, 2, 3, 4\}$ *e* $d_S(9) = 5$

Logo, o vetor $\delta_{\mathbb{N}_0} = (1, 1, 2, 2, 3, 3, 4, 4, 5, 5)$ é construído conforme o código acima.

Para compreender a complexidade desse algoritmo, note que a linha 5 do laço for irá percorrer o intervalo $[0, 3G]$, portanto a complexidade desse algoritmo na notação big-O é $O(G)$. Na linha 6, em cada etapa do laço, adicionamos um valor no vetor $\delta_{\mathbb{N}_0}$. O valor a ser inserido é uma soma, a soma de dois valores assume a complexidade $O(\log(G))$ referente a quantidade de bits utilizada para obter os valores. Como o valor de x está compreendido no intervalo $[0, 3G]$, a complexidade do algoritmo é de $O(G \times \log(G))$.

3.3.3 Ramificações

Este algoritmo está diretamente relacionado com a Proposição 3.2 e a construção de um semigrupo a partir de δ_S:

Código 4 – Código Python que gera as ramificações dos nós das árvores.

```
1  def Filho(S, x, G):
2      c = x + 1
3      g = S.g + 1
4  
5      if x > S.m:
6          m = S.m
7      else:
8          m = S.m + 1
9  
10     Sx = NumericalSemigroup(c, g, m)
```

```
11      Sx.vetor_decomposicao = [*S.vetor_decomposicao]
12      for y in range(x, 3*G + 1):
13          if S.vetor_decomposicao[y - x] > 0:
14              Sx.vetor_decomposicao[y] = S.vetor_decomposicao[y] - 1
15
16      return Sx
```

Fonte: O autor

Os parâmetros desta função são basicamente o semigrupo numérico, o elemento irredutível x que é o elemento chave para criação do novo semigrupo numérico, e também o valor de G solicitado. Basicamente para cada elemento irredutível utilizado em x o algoritmo acima cria uma cópia de δ_S de tal forma que a decomposição numérica é refeita considerando a retirada do elemento minimal irredutível dado conforme mostrado nas linhas 13 e 14.

Para entender melhor o exemplo abaixo, torna-se necessário o conhecimento do que cada elemento em δ_S representa. Os elementos que compõe o vetor de decomposição são gerados utilizando a Proposição 3.2 e da cardinalidade da decomposição numérica. Basicamente, cada elemento recebe a quantidade de maneiras que um elemento pode ser escrito como a soma de dois elementos em S. Por exemplo, quando $d_S(n) = 0$ sabemos que o elemento $n \notin S$. Quando $d_S(n) = 1$ temos que n é um elemento irredutível de S e consequentemente bastante importante para o contexto dos algoritmos apresentados. Quando $d_S(n) \geq 2$ dizemos que $n \in S$ e que pode ser escrito como a soma de outros elementos presentes em S. As Proposições 3.1 e 3.2 abordam essa questão com maior riqueza de detalhes.

Para o exemplo a seguir, torna-se necessário elucidar acerca da notação $\delta_S[x]$ em que x é um elemento no intervalo $[0, 3G]$, correspondendo a posição do elemento no vetor de decomposição numérica δ_S, iniciando-se na posição 0. Conforme observado na Proposição 3.1, quando $\delta_S[x] = 1$, tem-se um elemento irredutível. Os elementos irredutíveis são os elementos responsáveis por gerar de novos semigrupos numéricos.

Exemplo 3.5. Seja $\delta_S = (1, 0, 1, 1, 2, 2, 3, 3, 4, 4)$ o vetor de decomposição de $\{0, 2, \rightarrow\}$. Os elementos irredutíveis estão nas posições 2 e 3, pois $\delta_S[2] = \delta_S[3] = 1$, e irão gerar novos semigrupos numéricos.

- Para $\delta_S[2] = 1$:

  Usando a Proposição 3.2, encontramos $\delta_{S^x} = (1, 0, 0, 1, 1, 1, 2, 2, 3, 3)$, com $x = 2$. Daí $c(S^x) = 3, m(S^x) = 3, g(S^x) = 2$.

- Para $\delta_S[3] = 1$:

Usando a Proposição 3.2, encontramos $\delta_{S^x} = (1,0,1,0,2,1,2,2,3,3)$, com $x = 3$. Daí, $c(S^x) = 4, m(S^x) = 2$ *e* $g(S^x) = 2$.

Note que o exemplo acima descreve como o semigrupo numérico de $g = 1$ é utilizado para gerar os semigrupos numéricos de $g = 2$ a partir do vetor de decomposição.

Exemplo 3.6. Seja $\delta_S = (1,0,0,1,1,1,2,2,3,3)$ o vetor decomposição de $\{0,3,4,5,\rightarrow\}$. Os elementos irredutíveis estão nas posições 3, 4 e 5, pois $\delta_S[3] = \delta_S[4] = \delta_S[5] = 1$, e irão gerar novos semigrupos numéricos. Utilizaremos novamente a Proposição 3.2.

- Para $\delta_S[3] = 1$:

  Teremos um semigrupo numérico filho dado por $\delta_{S^x} = (1,0,0,0,1,1,1,1,2,2)$, em que, $c(S^x) = 4, m(S^x) = 4, g(S^x) = 3$.

  Para calcular δ_{S^x}, utiliza-se a Proposição 3.2.

- Para $\delta_S[4] = 1$:

  Teremos um semigrupo numérico filho dado por $\delta_{S^x} = (1,0,0,1,0,1,2,1,2,2)$, em que $x = 4$ e $c(S^x) = 5, m(S^x) = 3$ *e* $g(S^x) = 3$.

- Para $\delta_S[5] = 1$:

  Teremos um semigrupo numérico filho dado por $\delta_{S^x} = (1,0,0,1,0,1,2,2,2,2)$, em que $x = 5$ e $c(S^x) = 6, m(S^x) = 3$ *e* $g(S^x) = 3$.

Sobre a complexidade do algoritmo, temos o seguinte: conforme visto anteriormente, na Proposição 1.1 e na Proposição 1.3 temos que o elemento irredutível $x \leq 3G$, $c \leq 2G$ e $m \leq G + 1$, com isso teremos que a complexidade que compreende as linhas de 2 a 8 do algoritmo possuem complexidade $O(\log(G))$, em que $\log G$ representa a quantidade de bits necessária para realizar as operações. O laço *for* da linha 12 necessita de $O(G)$ passos e cada operação do laço ocorrerá em $\log(G)$. Portanto, o algoritmo possui a complexidade de tempo em $O(G \times \log(G))$.

3.3.4 Construção da árvore

A partir de todos os algoritmos anteriormente citados, temos a função principal, responsável por construir e salvar as contagens dos semigrupos por seus respectivos gêneros.

Código 5 – Código Python para exploração da travessia da árvore.

```
1 def Explore(G):
2     stack = []
3     raiz = NumericalSemigroup(0,0,1, G)
```

```
4     stack.append(raiz)
5     ngs = (G+1)*[0]
6
7     while stack:
8         S = stack.pop()
9         ngs[S.g] = ngs[S.g] + 1
10
11        if S.g < G:
12            for x in range(max(S.c, 1), S.c + S.m + 1 ):
13                if S.vetor_decomposicao_pos(x) == 1:
14                    stack.append(Filho(S, x, G))
15
16    return ngs
```

Fonte: O autor

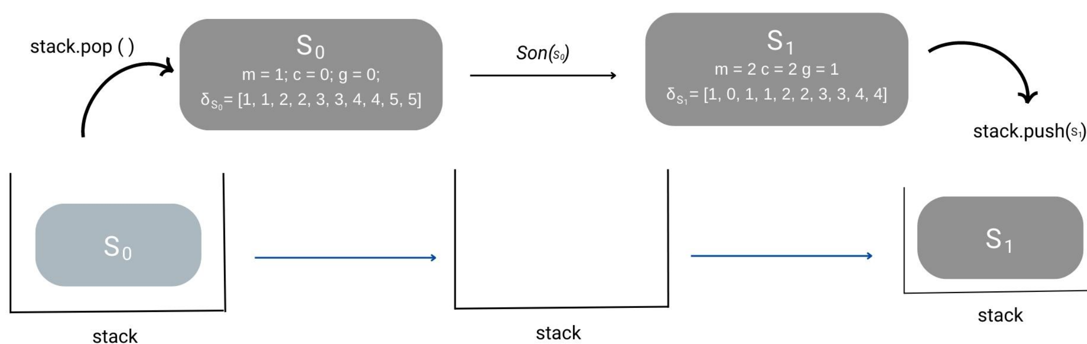
O Código 5 acima é o núcleo da construção da árvore. Na linha 4 utilizamos a função raiz apresentada anteriormente e armazenamos na pilha, sendo assim, o primeiro elemento da pilha. Este algoritmo possui a responsabilidade de gerar e armazenar um novo semigrupo numérico (linhas 12 à 14) a partir do semigrupo numérico retirado do topo da pilha na linha 8.

O raciocínio deste trecho de código é similar a DFS, na busca por profundidade cada nó vizinho é visitado e a cada iteração vai se aprofundando cada vez na árvore. A partir de um nó, visitamos o vizinho deste nó, que por sua vez, torna-se a referência para os próximos passos. Portanto, na DFS iremos percorrer o vizinho do vizinho e assim sucessivamente, quando o nó não apresenta mais um caminho, então retorna para outro vizinho do nó inicial e repete o procedimento até percorrer toda estrutura.

Exemplo 3.7. Para apresentar as ideias expressas nos algoritmos apresentados acima, podemos exemplificar o comportamento do algoritmo quando solicitamos $G = 3$. O primeiro passo é instanciar a pilha e adicionar o semigrupo raiz.

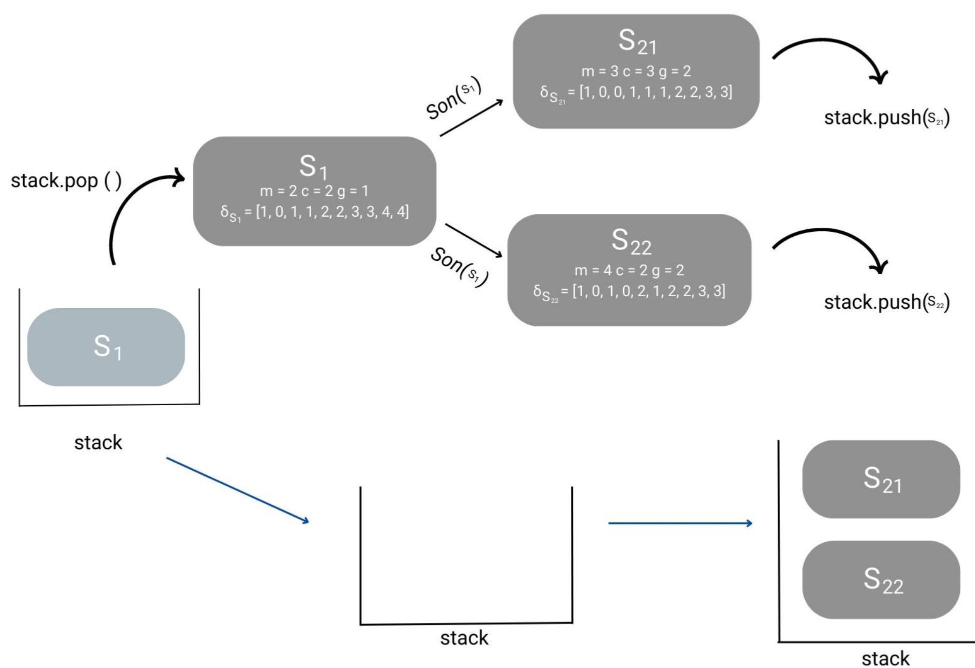
Na Figura 8 vemos que assim que a pilha entra no loop da linha 7, o topo da pilha S_0 é retirado para então utilizá-lo como referência para assim criar o semigrupo do próximo nível. O semigrupo S_0 irá gerar um novo objeto semigrupo numérico, a partir do vetor δ_{S_0}. Este novo objeto S_1 será adicionado no topo da pilha. Vale a pena ressaltar que a cada *stack.pop*() a contagem de semigrupos é realizada, incrementando 1 no seu respectivo valor g conforme visto na linha 11 do algoritmo acima. Note que as linhas de sucessão referentes a estrutura *stack* denotam paralelamente o estado da pilha durante o processo de retirada e inserção dos semigrupos. Como no final do processo da Figura 8, a pilha ainda não está vazia, daremos continuidade ao algoritmo.

Figura 8 – Execução do Algoritmo - Parte 1



Fonte: O autor

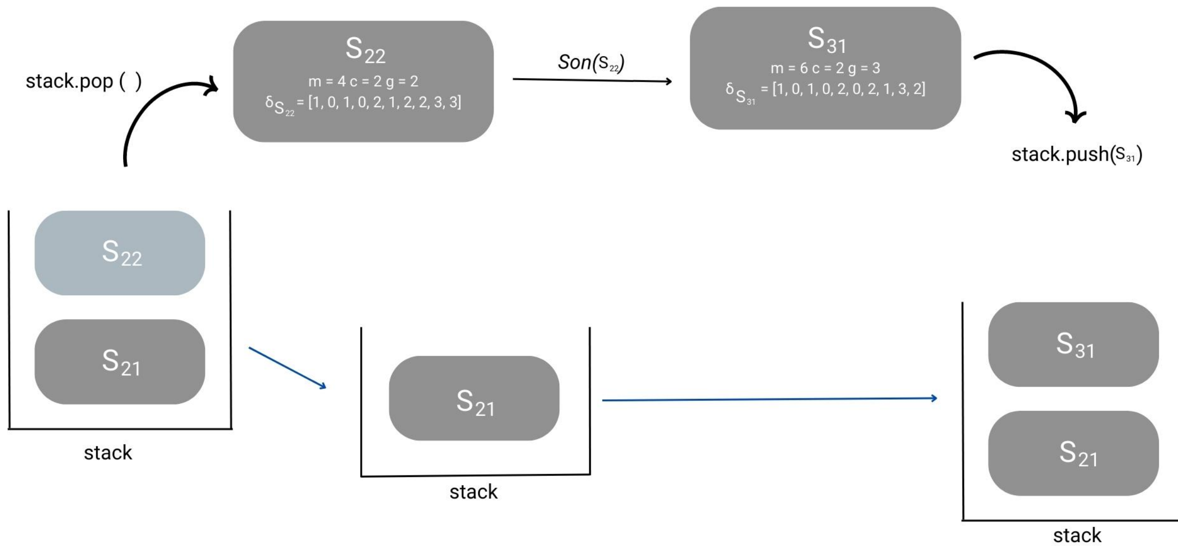
Figura 9 – Execução do Algoritmo - Parte 2



Fonte: O autor

Conforme pode ser observado na Figura 9, o semigrupo numérico S_1 é capaz de gerar dois filhos, S_{21} *e* S_{22} respectivamente. Ambos os objetos são inseridos na estrutura e serão utilizados conforme a Figura 10.

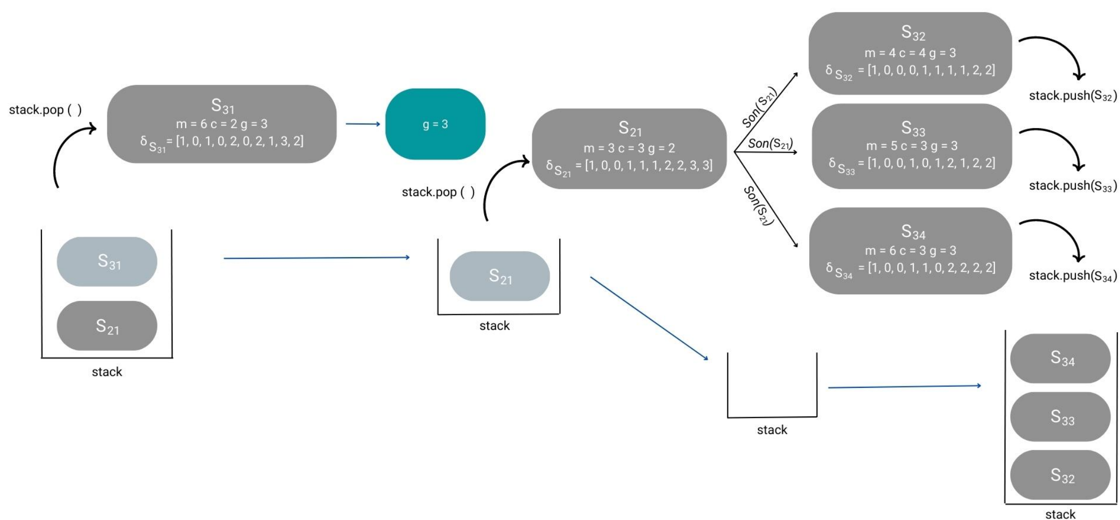
Figura 10 – Execução do Algoritmo - Parte 3



Fonte: O autor

Agora S_{22} é utilizado como referência para gerar um novo objeto S_{31}, que é consequentemente armazenado em nossa estrutura. E assim o próximo elemento a ser retirado da nossa estrutura será o último elemento que acabou de ser inserido, consequentemente S_{31}.

Figura 11 – Execução do Algoritmo - Parte 4



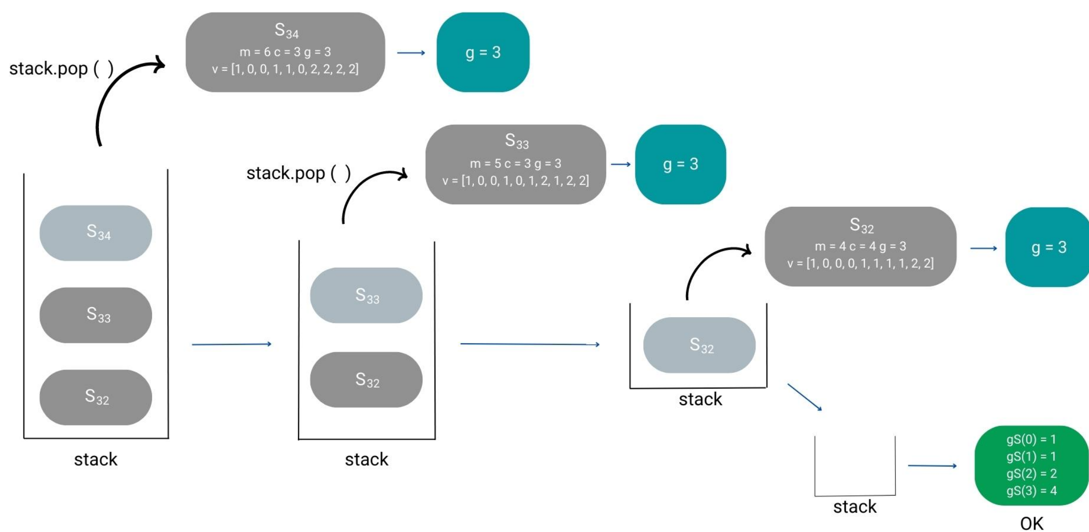
Fonte: O autor

Note que, ao retirar o elemento S_{31} já encontramos o primeiro semigrupo numérico tem gênero 3. Com isso, a linha 12 não permite que S_{31} gere novos objetos e armazene-os.

Portanto, o próximo passo é retirar outro elemento do topo da pilha, teremos agora S_{21} como referência, o qual é o segundo filho armazenado do semigrupo S_1. Nota-se aqui a característica da DFS em que aprofundamos a árvore na parte mais à direita percorrendo assim a sequência $\langle \mathbf{2}, \mathbf{3} \rangle$, $\langle 2, \mathbf{5} \rangle$, $\langle 2, \mathbf{7} \rangle$ após alcançarmos o nosso $g = 3$ a estrutura realizou um *backtrack* e irá aprofundar mais a partir do outro filho não visitado, no caso S_{21}. Por sua vez, na Figura 11, vemos que S_{21} acarretará três novos semigrupos armazenados na pilha para realização da contagem.

Agora iremos para a última parte da execução.

Figura 12 – Execução do Algoritmo - Final



Fonte: O autor

Como ilustrado na Figura 12, os elementos armazenados são prontamente utilizados para contagem dos semigrupos, pois já encontramos todos os semigrupos quando $g = 3$. E por fim, quando a pilha se esvazia, ocorre o encerramento da execução e os semigrupos foram contabilizados e salvos em seus respectivos gêneros.

3.3.5 Análise de complexidade do algoritmo

Para compreender a complexidade de tempo do algoritmo de exploração da árvore, note que na linha 14, o algoritmo realiza a chamada da função *Filho* no Código 4, para gerar cada semigrupo numérico da árvore de gênero G. Como foi discutido anteriormente, sabemos que a complexidade do algoritmo *Filho* é $O(G \times \log(G))$ e que a função é chamada para cada semigrupo numérico presente na árvore. A quantidade de semigrupos numéricos que constituem a árvore é dada por $\sum_{g=0}^{G} n_g$. Em Bras-Amorós (2009) foi provado que $n_g \leq 1+3 \cdot 2^{g-3}$, para todo $g \geq 3$. Dessa forma, $\sum_{g=0}^{G} n_g = O(2^G)$. Portanto, a complexidade de tempo para o algoritmo de construção da árvore é de $O(2^G \times G \times \log(G))$.

Já a complexidade de espaço que o algoritmo utiliza está ligada as propriedades de construção da árvore. O algoritmo de exploração da árvore é a busca em profundidade(DFS). Fromentin e Hivert (2016) declararam duas propriedades de como a pilha é preenchida durante a execução do algoritmo, sendo elas:

- À medida que se percorre a pilha da sua base até o topo, o gênero é incrementado;
- Para todo $g \in [0, G]$, cada semigrupo numérico de gênero g, que está provisoriamente na pilha, possui o mesmo pai. Conforme pode ser observado na Figura 11 onde os elementos empilhados são ramificações de um mesmo semigrupo.

Sabemos que, cada gerador da árvore de semigrupos, é criado a partir dos elementos irredutíveis no conjunto de $\{c(S), \ldots, c(S) + m(S) - 1\}$. Logo, a partir da Proposição 1.1 um semigrupo de gênero g terá no máximo $g + 1$ filhos. Portanto, a pilha conterá no máximo $g + 1$ semigrupos numéricos de gênero $g + 1$ para todo $g \leq G$. O tamanho da pilha está limitado a

$$M = \sum_{g=0}^{G} g = \frac{G(G+1)}{2}$$

Cada instância de um semigrupo numérico S, é representada computacionalmente por $c(S), g(S), m(S), \delta_S$. A partir da Proposição 1.1 temos que $c \leq 2g$, $m \leq g + 1$ e que o gênero varia conforme o G solicitado, portanto $g \leq G$. Estes inteiros requerem em memória um espaço de $O(\log(G))$ referente a quantidade de bits necessária para armazenamento. Já o $\delta_S = (d_S(0), \ldots, d_S(3G))$, cada valor do vetor requer $O(\log(G))$ em espaço de memória. Portanto, a complexidade de δ_S é $O(G \times \log(G))$. Isso conduz que o espaço de complexidade do algoritmo de exploração da árvore é:

$$O(M \times G \times \log(G)) = O(G^3 \times \log(G)).$$

3.4 Algoritmos em C++

Após utilizarmos a linguagem Python, foram necessárias realizar algumas melhorias para melhorar o desempenho do algoritmo. Devido ao Python possuir uma abstração em relação aos tipos de dados e também ao seu desempenho inferior em relação à linguagem C, para de fato realizar algumas melhorias no desempenho, o ideal foi realizar uma migração para C++ e automaticamente melhorar o desempenho da solução.

Embora a estrutura e a concepção do código tenham permanecidas as mesmas, a migração de Python para C++ trouxe consigo algumas melhorias significativas. Essas

mudanças resultaram em um desempenho aprimorado e em uma maior eficiência computacional, permitindo que o programa processasse grandes quantidades de dados com maior rapidez. Além disso, a transição para C++ aumentou a capacidade de personalização e a flexibilidade do código, facilitando a integração de novos recursos e funcionalidades. No geral, a migração para C++ foi um passo importante para aprimorar a qualidade e a robustez do algoritmo.

3.4.1 Instância de um semigrupo numérico

Código 6 – Código C++ da classe de Semigrupos Numéricos.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  class NumericalSemigroup {
4      private:
5          int c;
6          int g;
7          int m;
8          vector<__uint8_t> decs;
9          void Root(int G) {
10             for (int x = 0; x <= 3 * G; x++)
11                 this->decs.push_back(1 + floor(x / 2));
12         }
13     public:
14         NumericalSemigroup(int condutor, int genero,  int multiplicidade,
15                            int G = 0) {
16             this->c = condutor;
17             this->g = genero;
18             this->m = multiplicidade;
19             this->Root(G);
20         }
21
22         int getC() const {return c;}
23         int getG() const {return g;}
24         int getM() const {return m;}
25         __uint8_t* getVetorDecomposicaoPos0() {return &decs[0];}
26         __uint8_t* getVetorDecomposicaoPos(int x) {return &decs[x];}
27         const std::vector<__uint8_t>& getVetorDecomposicao() const
28           {return decs;}
29         void copy_decomposicao(vector<__uint8_t> decs){this->decs = decs;}
30         int getDecsx(int x){return this->decs[x];}
31         NumericalSemigroup getChild(NumericalSemigroup S, int x, int G)
32           {return child(S,x,G);}
33 };
```

Fonte: O autor

Além da melhora apenas pela troca de ferramenta, a linguagem C++ permite definir especificamente o tipo de dado a ser estruturado, com isto ao invés de utilizarmos um inteiro com 4 *bytes* podemos utilizar um vetor que armazene valores de até 1 byte. Conforme visto anteriormente em Fromentin e Hivert (2016), temos que os elementos de δ_S obedecem a seguinte propriedade $d_S(x) \leq 1 + \left\lfloor \frac{x}{2} \right\rfloor$. Logo, para $x \leq 510$ teríamos elementos em δ_S menores que 256, que podem ser representados em um espaço de 1 byte de memória, o que consequentemente auxilia no desempenho da solução.

3.4.2 Raiz e ramificações da árvore

O Código 7 é um método privado, semelhante ao que observamos na linguagem inicialmente utilizada no Código 3, não houve mudanças significativas, agora apenas a inserção dos elementos está padronizada para $uint8$.

Código 7 – Código C++ para atribuição do vetor de decomposição da raiz da árvore.

```
1 void Root(int G) {
2     for (int x = 0; x <= 3 * G; x++)
3         this->decs.push_back(1 + floor(x / 2));
```

Fonte: O autor

Uma das melhorias citadas por Fromentin e Hivert (2016) é a utilização de cursores para percorrer os endereços do vetor ao mesmo tempo, e acessar os valores com mais eficiência e velocidade utilizando tecnologias e instruções MMX e SSE do processador. Vale também ressaltar que o vetor armazenado esteja em alinhamento na memória, o que permite um acesso mais eficiente à memória.

Código 8 – Código C++ utilizado para gerar novas ramificações da árvore.

```
1 NumericalSemigroup child(NumericalSemigroup S, int x, int G){
2     int c = x + 1;
3     int g = S.getG() + 1;
4     int m;
5     x > S.getM() ? m = S.getM() : m = S.getM() + 1;
6
7     NumericalSemigroup Sx = NumericalSemigroup(c, g, m);
8     Sx.copy_decomposicao(S.getVetorDecomposicao());
9
10    int i = 0;
11    __uint8_t *src = S.getVetorDecomposicaoPos0();
12    __uint8_t *dst = Sx.getVetorDecomposicaoPos(x);
13
```

```
14      while (i <= 3*G - x){
15          if (*src > 0){
16              *dst = *dst - 1;
17          }
18          src++;
19          dst++;
20          i++;
21      }
22      return Sx;
23  }
```

Fonte: O autor

Nesta nova versão, nota-se que os cursores *src* e *dst* percorrem ao mesmo tempo, e os valores a serem acessados e modificados apenas necessitam das variáveis *src* e *dst*. A CPU utilizada permitiu a utilização de tecnologias **MMX** e **SSE**, o que permitiu o processamento em maior velocidade.

3.4.3 Exploração da árvore

Essa é a parte central do algoritmo, responsável pela chamada das outras funções e é responsável por iniciar e armazenar as informações da árvore mediante o G solicitado. Observe que o código é semelhante ao código em Python, mudando apenas a questão da sintaxe entre as linguagens.

Código 9 – Código C++ para exploração e travessia da árvore de semigrupos numéricos.

```
1   void Explore(int G){
2       stack<NumericalSemigroup>stack;
3       NumericalSemigroup root = NumericalSemigroup(0,0,1,G);
4       stack.push(root);
5
6       while (stack.size() != 0){
7           NumericalSemigroup S = stack.top();
8           stack.pop();
9           ngs[S.getG()] = ngs[S.getG()] + 1;
10
11          if (S.getG() < G){
12              for (int x = max(S.getC(), 1); x < S.getC() + S.getM() + 1; x++){
13                  if (*S.getVetorDecomposicaoPos(x) == 1){
14                      stack.push(S.getChild(S,x,G));
15                  }
16              }
17          }
```

```
18      }
```

Fonte: O autor

A execução do Exemplo 3.7 também aplica-se aqui, cuja finalidade é obter a contagem de semigrupos numéricos por gênero. No próximo tópico ficará perceptível a motivação em realizar o mesmo algoritmo em outra plataforma.

3.5 Otimizações

Além da otimização adquirida a partir da mudança de linguagem no algoritmo apresentado, ainda existem otimizações que podem ser utilizadas no algoritmo para melhorar o tempo de execução. Técnicas de vetorização, paralelização dos dados e alinhamento de memória permitiram uma maior velocidade para exploração da árvore.

3.5.1 SIMD, SSE e MMX

SIMD é a sigla para "Single Instruction, Multiple Data", que em português significa "Instrução Única, Dados Múltiplos". É uma técnica de programação que explora o paralelismo de dados em arquiteturas de processadores para executar operações em paralelo e em conjuntos de dados.

- **SSE** (Streaming SIMD Extensions): SSE é uma extensão que adiciona um conjunto de instruções SIMD ao conjunto de instruções básico do processador. Ela permite que operações matemáticas sejam executadas em paralelo em vários elementos de dados, melhorando o desempenho em certas tarefas, como processamento de imagem, processamento de áudio e vídeo, entre outros.

- **MMX** (MultiMedia eXtensions) MMX é outra extensão que fornece instruções para melhorar o desempenho em operações. Ele permite que operações sejam realizadas em paralelo em múltiplos elementos de dados, utilizando registros especiais chamados de registros MMX.

Ambas as tecnologias, SSE e MMX visam melhorar o desempenho de aplicações que fazem uso intensivo de cálculos matemáticos e processamento de dados, permitindo a execução paralela de instruções em diferentes elementos de dados. No entanto, é importante notar que SSE é uma tecnologia mais recente e avançada do que MMX, fornecendo um conjunto mais amplo de instruções e recursos para otimização de desempenho.

3.5.2 Vetorização

Essa técnica envolve a reescrita de código para realizar operações em vetores ou matrizes em vez de processar cada elemento individualmente.

A ideia por trás da vetorização é explorar a capacidade dos processadores de executar instruções **SIMD**, que operam em paralelo em múltiplos elementos de dados. Em vez de realizar operações em um único elemento de cada vez, a vetorização permite realizar a mesma operação em vários elementos simultaneamente.

As CPUs costumam trabalhar com 8 bytes ou 16 bytes utilizando vetores de extensão. Tendo em vista estes aspectos e as considerações apresentadas a classe de semigrupos numéricos anteriormente apresentada foi modificada.

Código 10 – Código C++ para classe de semigrupos numéricos com otimizações.

```
1  typedef uint_fast64_t ind_t;
2  typedef uint8_t epi8 __attribute__ ((vector_size (16)));
3  typedef uint8_t dec_numbers[SIZE] __attribute__ ((aligned (16)));
4  typedef epi8 dec_blocks[NBLOCKS];
5
6  class alignas(16) NumericalSemigroup {
7      public:
8          ind_t conductor;
9          ind_t genus;
10         ind_t multiplicity;
11
12         union{
13             dec_numbers decs;
14             dec_blocks blocks;
15         };
16 };
```

Fonte: O autor

O $uint_fast64_t$ é um tipo de dado inteiro sem sinal de 64 bits com largura de pelo menos 64 bits. Esse tipo de dado é usado para representar números inteiros sem sinal que exigem pelo menos 64 bits de armazenamento e é projetado para fornecer um acesso rápido e eficiente a esses valores. A vantagem é que este tipo de dado oferece um equilíbrio entre tamanho e desempenho e permite que a implementação escolha um tipo que ofereça melhor desempenho em uma determinada plataforma.

Para alinhar os elementos da classe NumericalSemigroup na memória, utiliza-se a palavra-chave *alignas* antes da definição da classe. Isso garantirá que os objetos da classe sejam armazenados em posições de memória que respeitem o alinhamento de 16 bytes.

A vetorização foi aplicada no vetor de decomposição que agora representa um vetor de tamanho *SIZE* onde cada elemento possui 8 bits e que deve ser alinhado em uma fronteira de 16 bytes na memória. Isso significa que o endereço de memória em que o vetor começa deve ser múltiplo de 16. O alinhamento de memória é uma técnica usada para otimizar o acesso a dados em arquiteturas de computador.

Além disso, o vetor de decomposição foi dividido em *NBLOCKS* de 16 bytes. O motivo dessa abordagem é que no algoritmo *Filho*, responsável por gerar as ramificações dos semigrupos numéricos é necessário realizar a cópia do semigrupo numérico inicial. Ao invés de uma atribuição direta, torna-se mais eficiente a utilização do Código 11.

Código 11 – Código C++ para cópia do vetor de decomposição em blocos.

```
1 void copy_blocks(dec_blocks &dst, dec_blocks const &src) {
2         for (ind_t i=0; i<NBLOCKS; i++) dst[i] = src[i];
3     }
```

Fonte: O autor

Ao copiar um vetor em blocos, aproveita-se das instruções SIMD para processar vários elementos de uma só vez. Em vez de copiar elemento por elemento, realiza-se a cópia dos blocos de dados em uma única operação, reduzindo a quantidade de instruções necessárias e aumentando a eficiência do processo de cópia. Copiar em blocos também pode aproveitar o cache do processador de maneira mais eficiente. Ao realizar a cópia em blocos contíguos reduz o número de acessos à memória principal, aproveitando o cache local do processador, que é mais rápido em termos de latência de acesso.

Considerando agora a nova abordagem referente ao vetor de decomposição o *loop while* do algoritmo *Filho* é convertido em um **loop for** responsável por atribuir o vetor de decomposição.

Código 12 – Código C++ para gerar o vetor de decomposição numérica da ramificação.

```
1 start_block = x >> 4; // Índice do bloco que contem o irredutível x
2 decal = gen & 0xF; // Deslocamento de x no bloco
3
4 for (auto i=start_block+1; i<NBLOCKS; i++){
5     block = load_unaligned_epi8(src.decs + ((i-start_block)<<4) - decal);
6     dst.blocks[i] -= ((block != zero) & block1);
7 }
```

Fonte: O autor

A instrução *load_unaligned_epi*8 é uma instrução específica da tecnologia SSE e é responsável por carregar um vetor de elementos de 8 bits não alinhados na memória em uma variável de 16 bytes. A comparação nos (block != zero) nos retorna um bloco que contém 0 nos elementos que são iguais a 0 e 255 nos elementos diferentes de 0. E o resultado recebe uma operação bitwise and-ed que permite então converter os elementos 255 em 1. A partir disso, executa-se a operação de subtração dos elementos conforme a Proposição 3.2.

Exemplo 3.8. Considere que $block = \{0, 6, 0, 7, 0, 7, 1, 8, 2, 8, 3, 9, 4, 9, 5, 10\}$, $zero = \{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$ e $block1 = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$ e $dst.blocks(i) = \{13, 12, 13, 13, 14, 14, 15, 15, 16, 16, 17, 17, 18, 18, 19, 19\}$

- Para block != zero, teremos como resultado: $\{0, 255, 0, 255, 0, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255\}$
- Considerando o resultado acima iremos aplicar o bitswise and, teremos então que o resultado de ((block != zero) block1): $\{0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$.
- Portanto, considerando o resultado acima a linha 6 do algoritmo irá subtrair 1 ou 0 em cada elemento de $dst.blocks(i)$, portanto teremos como resultado deste exemplo o seguinte bloco: $\{13, 11, 13, 12, 14, 13, 14, 14, 15, 15, 16, 16, 17, 17, 18, 18\}$. Este bloco por sua vez, faz parte do vetor de decomposição numérica do semigrupo numérico gerado.

O algoritmo segue este raciocínio para a atribuição dos vetores de decomposição e seus respectivos blocos de memória.

3.5.3 Paralelização

A próxima otimização envolve o uso de paralelismo na exploração da árvore. As CPUs presentes nos computadores atuais contam, em média, de 2 a 8 núcleos de processamento. Todavia, essa versão de algoritmo de exploração visa utilizar apenas um único núcleo, ou seja, apenas uma fração da capacidade máxima de uma CPU.

O princípio do algoritmo está em explorar diferentes ramos da árvore através dos diferentes núcleos do computador e de forma paralela. Por outro lado, é necessário garantir que todos os núcleos estejam ocupados para que uma nova ramificação seja criada quando um núcleo terminar de processar o anterior.

Para este fim, utiliza-se a tecnologia Cilk++, uma linguagem de uso geral projetada para computação paralela multithread. Há dois termos importantes quando se trata dos recursos paralelos do Cilk++.

- **cilk spawn**: É responsável por indicar que a chamada pode ser realizada em paralelo com o restante do código e em segurança.
- **cilk sync**: Tem por função alertar sobre a fila de execução, em casos em que o procedimento atual não pode prosseguir até que os anteriores tenham sido concluídos.

Código 13 – Código Cilk++ utilizado para travessia paralela da árvore.

```
1  void Explore(const SemigrupoNumerico s){
2      unsigned long int nbr = 0;
3
4      if (s.genero < MAX_GENUS - STACK_BOUND){
5
6          auto son = generator_iter<CHILDREN>(s);
7          while (son.move_next()){
8              cilk_spawn walk_children(remove_generator(s, son.get_gen()));
9              nbr++;
10         }
11         cilk_results[s.genero] += nbr;
12     }
13     else
14         Explore_stack(s, cilk_results.get_array());
15 }
```

Fonte: O autor

Nessa versão de código é possível perceber que, para que a *Explore Stack* seja usada mais de 99% do tempo, o valor ótimo de *STACK BOUND* estava em torno de 10 a 12 para o gênero no intervalo de [45,64], demonstrando que função recursiva do Cilk++ garante que o trabalho pode equilibrado entre os diferentes núcleos conforme sugerido por Fromentin e Hivert (2016). As demais otimizações e arquivos dos algoritmos podem ser encontrados no repositório Duarte (2022).

3.6 Tempos de execução

A tabela abaixo representa a relação do tempo médio necessário para compilar e executar o código desenvolvido para cada uma das linguagens de programação utilizadas.

Os resultados apresentados na Tabela 1 denotam que apenas pela mudança de linguagem já obtivemos uma melhora significativa a partir de $g = 30$. Também após a realização de mais alguns ajustes de otimização foi possível melhorar ainda mais o desempenho.

Tabela 1 – Tempos de Execução (em segundos)

g	Python	C++	C++ com Otimização	Cilk++
10	0.054	0.004	0.002	-
15	0.076	0.016	0.015	0.001
20	0.058	0.108	0.096	0.002
25	7.809	1.040	0.846	0.008
30	111.143	12.988	10.910	0.061
33	525.600	58.860	48.099	0.207
35	-	166.812	130.320	0.438
38	-	762.578	583.961	1.624
41	-	3622.194	3072.124	7.320
45	-	-	-	51.290
50	-	-	-	676.9
55	-	-	-	4166
60	-	-	-	49190
61	-	-	-	80680
62	-	-	-	129300
63	-	-	-	211000
64	-	-	-	339400

Fonte: O autor

Em auxílio ao desenvolvimento deste trabalho utilizou-se o sistema de processamento por hospedagem em nuvem da Google, o Google Cloud. A combinação das máquinas virtuais e hospedagem no Google Cloud oferece aos desenvolvedores um ambiente flexível, poderoso e escalável para explorar árvores de semigrupos numéricos. Isso impulsiona a eficiência, o desempenho e a capacidade de lidar com grandes volumes de dados, permitindo o avanço na pesquisa e no desenvolvimento nessa área. A máquina selecionada para os processos apresenta como configuração uma CPU AMD Milan, do tipo c2d-highcpu-8, com 8 núcleos de processamento e 2 threads por núcleo, a qual foi utilizada para calcular até o n_{64}.

3.7 Execução do algoritmo

Para garantir a execução adequada do algoritmo, é necessário ter um ambiente com o sistema operacional Ubuntu 16 e o compilador Cilk++ instalados, juntamente com os arquivos específicos do algoritmo. No entanto, enfrentamos alguns desafios ao instalar o Cilk++ em diferentes máquinas virtuais e para garantir a portabilidade do algoritmo, optamos por criar uma imagem Docker e hospedá-la no Docker Hub.

Ao disponibilizar essa imagem no Docker Hub, torna-se mais fácil compartilhar e

Tabela 2 – Alguns valores de n_g

g	n_g	g	n_g	g	n_g	g	n_g
0	1	21	62 194	42	2 058 356 522	63	54 749 244 915 730
1	1	22	103 246	43	3 353 191 846	**64**	**88 754 191 073 328**
2	2	23	170 963	44	5 460 401 576	65	143 863 484 925 550
3	4	24	282 828	45	8 888 486 816	66	233 166 577 125 714
4	7	25	467 224	46	14 463 633 648	67	377 866 907 506 273
5	12	26	770 832	47	23 527 845 502	68	612 309 308 257 800
6	23	27	1 270 267	48	38 260 496 374	69	992 121 118 414 851
7	39	28	2 091 030	49	62 200 036 752	70	1 607 394 814 170 158
8	67	29	3 437 839	50	101 090 300 128	71	2 604 033 182 682 582
9	118	30	5 646 773	51	164 253 200 784	72	4 218 309 716 540 814
10	204	31	9 266 788	52	266 815 155 103		
11	343	32	15 195 070	53	433 317 458 741		
12	592	33	24 896 206	54	703 569 992 121		
13	1 001	34	40 761 087	55	1 142 140 736 859		
14	1 693	35	66 687 201	56	1 853 737 832 107		
15	2 857	36	109 032 500	57	3 008 140 981 820		
16	4 806	37	178 158 289	58	4 880 606 790 010		
17	8 045	38	290 939 807	59	7 917 344 087 695		
18	13 467	39	474 851 445	60	12 841 603 251 351		
19	22 464	40	774 614 284	61	20 825 558 002 053		
20	37 396	41	1 262 992 840	62	33 768 763 536 686		

Fonte: O autor

distribuir o ambiente de execução do algoritmo. Isso permite que outras pessoas possam acessar e utilizar a imagem em seus próprios computadores, eliminando a necessidade de configurar manualmente o ambiente em cada máquina. A imagem está disponibilizada em um repositório no Docker Hub em Duarte (2023).

As flags utilizadas para compilar o código-fonte são opções passadas para o compilador, neste caso utilizamos o compilador *g++-5*, para definir certas configurações e comportamentos durante a compilação. Sendo elas

- **-std=c++11**: Define que o código será compilado usando o padrão C++11, permitindo o uso de recursos mais recentes da linguagem.

- **-fcilkplus**: Ativa o suporte ao Cilk++, uma extensão do C++ que permite a programação paralela utilizando a biblioteca Cilk.

- **-g**: Inclui informações de depuração no executável gerado, facilitando a identificação de erros e bugs durante a execução.

- **-Wall**: Ativa o nível máximo de warnings (avisos) durante a compilação, ajudando a identificar possíveis problemas no código.
- **-O3**: Ativa a otimização de nível 3, que realiza várias otimizações no código para melhorar o desempenho.
- **-DMAX_GENUS=64**: Define a macro MAX_GENUS com o valor 64. Representando o nível G solicitado para construção da árvore.
- **-march=native**: Gera um código otimizado para a arquitetura do computador em que a compilação está sendo feita.
- **-mtune=native**: Otimiza o código para a arquitetura específica do processador.
- **-c**: Indica que o compilador deve apenas gerar o código objeto, sem fazer a ligação com outros módulos.

As flags utilizadas são essenciais para compilar corretamente o código e obter um executável com alto desempenho. Cada flag possui uma função específica e contribuem para a eficiência e qualidade da execução do algoritmo.

4 Considerações Finais

Neste TCC, exploramos os conceitos, propriedades e exemplos relacionados aos semigrupos numéricos. Aborda-se também a origem e o contexto do problema apresentado, construindo assim uma base sólida para compreensão do principal objeto de estudo deste trabalho, que é a árvore de semigrupos numéricos.

Abordamos a construção dos semigrupos numéricos através da árvore, a qual é a maneira de se explorar a contagem de semigrupos numéricos e obter os elementos da sequência (n_g). A partir da árvore também foi possível estabelecer uma conexão matemática e computacional, através da aplicação do algoritmo de busca em profundidade (DFS) para explorar a árvore de semigrupos numéricos.

No decorrer do trabalho foram apresentadas as estruturas e recursos computacionais necessários para criação do algoritmo e considerações acerca de sua complexidade de espaço e tempo. Permitindo assim, observar as possíveis melhorias e considerações práticas na implementação do algoritmo. Apresentam-se algumas otimizações no algoritmo e comparativos entre as versões dos algoritmos.

Importante ressaltar que o desenvolvimento contou com a busca de referências contextuais e teóricas citadas no trabalho *The right-generators descendant of a numerical semigroup* elaborado por Bras-Amorós e Fernández-González (2019). O trabalho pode ser acessado a partir do seguinte repositório no GitHub em Bras-Amorós e Fernández-González (2020).

Observou-se que os valores obtidos para n_{64} correspondem ao que já foi apresentado nos trabalhos de Bras-Amorós (2008) e Fromentin e Hivert (2016). Os valores obtidos da sequência corroboram com as conjecturas apresentadas por Brás-Amorós. Utilizando o n_{64} como referência, teremos:

1. $\frac{n_{64}}{n_{63}} \approx 1,6211$ que está próximo de $\frac{1+\sqrt{5}}{2}$;
2. $\frac{n_{63}+n_{62}}{n_{64}} \approx 0,9973$ que está próximo de 1;
3. $n_{62} + n_{63} = 88518008452416 \leq 88754191073328 = n_{64}$.

4.1 Trabalhos Futuros

Após consolidar a travessia da árvore, surgem oportunidades para otimizações e aprimoramentos. Algumas abordagens promissoras incluem investigar e implementar o uso de estruturas de dados especializadas que visam reduzir a complexidade e acelerar

as operações. Executar o algoritmo em uma máquina que possua maior quantidade de núcleos de processamento para obter maior velocidade na construção das ramificações da árvore. Além disso, técnicas de paralelização podem ser exploradas, como *threads*, processamento em GPU ou TPU.

A computação distribuída, envolvendo clusters ou sistemas de nuvem, também é uma opção para acelerar o processamento. Comparar o uso de diferentes ferramentas e bibliotecas otimizadas para paralelização, como OpenMP, CUDA e OpenCL, pode facilitar a implementação e alcançar melhor desempenho. Essas abordagens têm o potencial de tornar a construção da árvore de semigrupos numéricos mais eficiente.

Conclui-se que o estudo dos semigrupos numéricos e da árvore de semigrupos numéricos oferece oportunidades de trabalho tanto na área matemática quanto na área de computação.

Por fim, esperamos que este trabalho possa ser utilizado como um recurso para interessados em explorar e aplicar esses conceitos na área matemática e computacional acerca de semigrupos numéricos.

Referências

BRAS-AMORÓS, M. Fibonacci-like behavior of the number of numerical semigroups of a given genus. *Semigroup Forum*, v. 76, p. 379–384, 2008. Citado na página 63.

BRAS-AMORÓS, M. Bounds on the number of numerical semigroups of a given genus. *Journal of Pure and Applied Algebra*, v. 213, n. 6, p. 997–1001, 2009. ISSN 0022-4049. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0022404908002260>. Citado na página 49.

BRAS-AMORÓS, M.; FERNÁNDEZ-GONZÁLEZ, J. The right-generators descendant of a numerical semigroup. *Math. Comput.*, v. 89, p. 2017–2030, 2019. Citado na página 63.

BRAS-AMORÓS, M.; FERNÁNDEZ-GONZÁLEZ, J. *The right-generators descendant of a numerical semigroup*. 2020. Disponível em: <https://github.com/mbrasamoros/RGD-algorithm>. Citado na página 63.

CORMEN, T. H. et al. *Introduction to Algorithms, Third Edition*. 3rd. ed. [S.l.]: The MIT Press, 2009. ISBN 0262033844. Citado 2 vezes nas páginas 29 e 32.

DUARTE, B. *Árvore de Semigrupos Numéricos - TCC*. 2022. Disponível em: <https://github.com/Mexazonic/TCC-Semigrupos-Numericos>. Citado na página 58.

DUARTE, B. *Imagem DockerHub*. 2023. Disponível em: <https://hub.docker.com/repository/docker/mexazonic/tcc2/general>. Citado na página 60.

FROMENTIN, J.; HIVERT, F. Exploring the tree of numerical semigroups. *Mathematics of Computation*, American Mathematical Society, n. 85, p. 2553–2568, jan. 2016. 14 pages. Disponível em: <https://hal.science/hal-00823339>. Citado 13 vezes nas páginas 20, 26, 27, 35, 36, 37, 38, 39, 40, 50, 52, 58 e 63.

HALIM, S.; HALIM, F. *Competitive Programming 3: The New Lower Bound of Programming Contests*. Lulu.com, 2013. ISBN 9788392212355. Disponível em: <https://books.google.com.br/books?id=vUc-nwEACAAJ>. Citado na página 29.

JúNIOR, J. M. Contagem de semigrupos numéricos de mesmo gênero por meio de gapsets. *Dissertação (Mestrado Profissional em Matemática)*, p. 39 f., il., 2020. Citado 2 vezes nas páginas 23 e 35.

RODRIGUES, A. L. F. Semigrupos numéricos com multiplicidade fixada e proposta de atividade para o ensino médio com utilização do geogebra. *Dissertação (Mestrado Profissional em Matemática)*, p. 57 f., il., 2020. Citado na página 25.

ROSALES, J.; GARCÍA-SÁNCHEZ, P. *Numerical Semigroups*. New York, NY: Springer New York, 2009. ISBN 978-1-4419-0160-6. Citado na página 25.

SYLVESTER, J. *Mathematical questions with their solutions*. [S.l.: s.n.], 1884. v. 4-6. Citado na página 19.