Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Instituto de Matemática e Estatística

Juliana Macario de Souza

Sobre ferramentas para análise automatizada de algoritmos

Rio de Janeiro
2018

Juliana Macario de Souza

Sobre ferramentas para análise automatizada de algoritmos

Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Orientador: Prof. Dr. Fabiano de Souza Oliveira

Rio de Janeiro
2018

CATALOGAÇÃO NA FONTE

UERJ/REDE SIRIUS/BIBLIOTECA CTC/A

S729 Souza, Juliana Macario de.
Sobre ferramentas para análise automatizada de algoritmos. – 2018.
108f.: il.

Orientador: Fabiano de Souza Oliveira
Dissertação (Mestrado em Ciências Computacionais) - Universidade do Estado do Rio de Janeiro, Instituto de Matemática e Estatística.

1. Algoritmos - Teses. I. Oliveira, Fabiano de Souza. II. Universidade do Estado do Rio de Janeiro. Instituto de Matemática e Estatística. III.Título.

CDU 510.5

Patricia Bello Meijinhos CRB7/5217 - Bibliotecária responsável pela elaboração da ficha catalográfica

_____ _____

Assinatura Data

Juliana Macario de Souza

Sobre ferramentas para análise automatizada de algoritmos

Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Aprovada em 13 de setembro de 2018.

Banca Examinadora:

Prof. Dr. Fabiano de Souza Oliveira (Orientador)
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Paulo Eustáquio Duarte Pinto
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Valmir Carneiro Barbosa
Universidade Federal do Rio de Janeiro

Prof. Dr. Fábio Protti
Universidade Federal Fluminense

Rio de Janeiro
2018

AGRADECIMENTOS

Gostaria de expressar minha imensa gratidão a todos que de alguma forma colaboraram para a concretização deste trabalho. Destaco aqueles que foram indispensáveis para alcançar tal objetivo.

Agradeço a Deus, primeiramente pela vida. E também, por me conceder força, sabedoria e por atender os meus pedidos em todos os momentos.

Agradeço aos meus pais, Eliza e José Guilherme, primeiramente por me darem a vida, um lar harmonioso e uma infância inesquecível. Lembro-me que muito cedo minha mãe me ensinou a ler e escrever, sempre estimulando meu raciocínio através de palavras cruzadas e quebra-cabeças. Por fim, agradeço a eles por lutarem diariamente para me oferecer sempre o melhor que podem, por me apoiarem em todas as etapas dos meus estudos e por me mostrarem que esse é o caminho certo a seguir. Sem todo esse apoio este trabalho não seria possível.

Agradeço ao meu noivo Cássio por estar comigo desde o início da minha graduação, por sempre ter me motivado a fazer o mestrado e também pelo companheirismo, compreensão, dúvidas tiradas em trabalhos acadêmicos e especialmente pelo seu amor.

Agradeço as inesquecíveis professoras: Regina, que foi a minha primeira professora e aturou os meus choros diários devido ao bullying sofrido; Etany, que foi a responsável pela minha alfabetização definitiva; Mirtes, que me ensinou toda base do futuro, da 1ª a 4ª série. Todas as três sempre me motivaram a ir mais longe pois sempre acreditaram que eu teria um bom futuro.

Agradeço aos professores do Instituto Cylleno: Edson, Lenice e Cidália (em memória) e aos professores mais especiais do ISERJ: Magui, Telma, Luc, Marcelo e Cláudio, por marcarem minha vida, por serem os professores que mais acreditaram em mim, me incentivaram e me ajudaram a ingressar na universidade. Agradeço também a todos os outros professores que passaram pela minha vida e que contribuíram para minha formação, pois seria impossível nomear todos.

Agradeço ao meu orientador Fabiano por resgatar em sua primeira aula de algoritmos a minha paixão pela computação que havia se transformado em repúdio por motivos que ficaram no passado e por ter aceitado me orientar. Agradeço também pelas diversas vezes em que acreditou em mim até quando eu mesma duvidei. Por fim, agradeço pela paciência, dedicação e motivação. Um exemplo de professor e um orientador nota 10.

Agradeço a minha amiga Lívia, mais um presente deste mestrado, por estar comigo desde o processo seletivo, cursando todas as disciplinas e passando dias inteiros estudando comigo. Agradeço especialmente pela sua amizade, que levarei para vida.

Agradeço aos professores Paulo e Igor do CCOMP, por transmitirem seus conhecimentos com muita dedicação, sempre ajudando e motivando todos os alunos.

Agradeço aos professores da UFRJ: Valmir, pela contribuição na minha banca e pela parceria nos artigos; Luís Alfredo, pela amizade e motivação.

Agradeço ao Estado brasileiro pela oportunidade que me foi dada de usufruir de um ensino público de qualidade, mesmo com todas as dificuldades que o país sempre passou. Em particular, agradeço a FAPERJ pelo suporte financeiro concedido para a realização deste trabalho.

Agradeço a todos os meus amigos e familiares que foram pacientes nas minhas ausências devido aos estudos, pelo apoio e carinho.

Este é o meu mandamento: amai-vos uns aos outros, como eu vos amo. Ninguém tem maior amor do que aquele que dá a sua vida por seus amigos.

João 15:12-13

RESUMO

SOUZA, Juliana. *Sobre ferramentas para análise automatizada de algoritmos*. 2018. 108 f. Dissertação (Mestrado em Ciências Computacionais) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2018.

Este trabalho tem por objetivo comparar ferramentas que produzem análise automatizada de algoritmos, através da realização de experimentos com diversos tipos de algoritmos e estruturas de dados. Para cada um, avaliam-se qualitativamente os resultados obtidos por tais ferramentas. Além disso, tem-se por objetivo apresentar a metodologia e as funcionalidades da ferramenta *EMA (EMpirical Analysis of algorithms)*. Finalmente são apresentados os resultados de uma metodologia de análise empírica chamada *teste do Big-Enough*. Esta metodologia consiste em determinar o valor inicial de tamanho de entrada para o qual verifica-se empiricamente a complexidade analítica. A metodologia foi aplicada a diversos algoritmos, cujas funções de complexidade de tempo são de variadas classes.

Palavras-chave: Algoritmos. Complexidade de algoritmos. Análise empírica de algoritmos. Análise automática de algoritmos.

ABSTRACT

SOUZA, Juliana. *About tools for automated algorithm analysis*. 2018. 108 f. Dissertação (Mestrado em Ciências Computacionais) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2018.

This work aims to compare tools that produce an automated analysis of algorithms, through experiments with several types of algorithms and data structures. For each one, the results obtained by such tools are evaluated qualitatively. In addition, the objective is to present the methodology and functionalities of the tool *EMA (EMpirical Analysis of algorithms)*. Finally, the results of an empirical analysis methodology called *Big-Enough test* are presented. This methodology consists of determining the initial value of input size for which analytic complexity is verified empirically. The methodology was applied to several algorithms, whose time complexity functions are of several classes.

Keywords: Algorithms. Complexity of Algorithms. Empirical Analysis of Algorithms. Automatic Analysis of Algorithms.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

SUMÁRIO

INTRODUÇÃO

Problema

Em geral, uma característica muito importante que deve ser considerada em qualquer algoritmo é como seu tempo de execução varia com as diferentes entradas. O tempo de computação, dada uma certa entrada, depende de diversos fatores, dentre os quais estão o algoritmo sendo empregado, o código de máquina gerado pelo compilador da linguagem escolhida para implementar o algoritmo e a arquitetura do computador em que o programa será executado. A área de análise de algoritmos preocupa-se com a primeira variável, isto é, sendo as mesmas todas as outras variáveis, visa determinar como diferentes algoritmos impactam os respectivos tempos de execução. Em verdade, tal área preocupa-se, de maneira mais geral, com recursos requeridos pelo algoritmo, sendo os mais usuais o tempo, medido através do número de passos, e o espaço ocupado, medido através da quantidade de memória alocada. É possível empreender este estudo através de uma abordagem analítica, aplicando-se técnicas específicas de contagem de passos de execução do algoritmo (ou da contagem de outro recurso de interesse) a partir da descrição do mesmo. Também pode-se proceder tal análise por uma abordagem empírica, analisando-se a execução propriamente dita do algoritmo sob diversas entradas com o uso de ferramentas ou algoritmos específicos para medição dos recursos consumidos. Ambos os métodos visam determinar uma expressão matemática que descreve o comportamento do tempo de execução de um algoritmo, que é chamada *complexidade* do algoritmo. O cálculo da complexidade (seja ela de tempo, de espaço, ou de outro recurso) auxilia na escolha para uso prático de um dentre vários algoritmos que resolvem o mesmo problema. Embora a abordagem analítica seja preferível por sua natureza precisa, alguns algoritmos resistem a serem analisados por ela, devido à complexidade do tratamento matemático. Este é o caso, quase que usual, das análises de caso médio, úteis em aplicações práticas. Por outro lado, são poucas as ferramentas que utilizam a abordagem empírica com a geração da complexidade assintótica em função das variáveis de entrada. A abordagem comum é aquela de medir o consumo de recurso para uma dada entrada, deixando de fora a inferência da função mais provável que corresponde às medições observadas.

Objetivo

O objetivo deste trabalho é o estudo de ferramentas que produzem análise automatizada de complexidade de algoritmos, através da realização de experimentos com diversos tipos de algoritmos e estruturas de dados e comparando os resultados obtidos para cada um. Além disso, é objetivo apresentar a metodologia, funcionalidades e análise qualitativa de uma ferramenta chamada EMA (acrônimo de *EMpirical Analysis of algorithms*) [1], desenvolvida pelo orientador deste trabalho. O EMA visa através de uma abordagem empírica analisar algoritmos de maneira a distinguir das demais análises empíricas fundamentalmente nos seguintes quesitos: (i) na profundidade da automação fornecida pela ferramenta em relação às demais, exigindo do usuário apenas as informações estritamente

necessárias, e (ii) geração do produto da abordagem analítica, que é a obtenção do comportamento assintótico dos algoritmos. Neste sentido, parte do objetivo com todas as implementações a serem feitas e analisadas com o EMA é de contribuir com um teste compreensivo da ferramenta, através da análise de algoritmos de diversas classes de complexidade, servindo de base para o eventual refinamento do EMA. Também é objetivo apresentar uma metodologia, a qual chamaremos de *teste do Big-Enough*, que visa obter o menor tamanho de entrada para o qual verifica-se empiricamente a complexidade analítica.

Justificativa

A análise empírica de algoritmos é de aplicação conveniente em diversos cenários. Dentre eles, podemos citar (i) o uso didático para verificar se o método analítico foi empregado corretamente fazendo-se a comparação entre o resultado esperado e o medido na prática; (ii) quando não se tem acesso ao código-fonte com a implementação de um algoritmo e ainda assim é desejável determinar sua complexidade; (iii) quando a complexidade é desconhecida analiticamente, pela dificuldade da matemática envolvida no algoritmo específico; (iv) auxiliar na escolha do algoritmo que possui função com menor constante multiplicativa quando há mais de um algoritmo de mesma complexidade e (v) quando é necessário prever a quantidade de tempo/memória necessários para execução do algoritmo para entradas as quais o algoritmo ainda não foi submetido e deseja-se prever uma quantidade numérica aproximada de qual será o consumo de recursos, observando-se tais consumos para entradas já utilizadas em execuções anteriores.

Para todas essas situações, é de grande importância uma ferramenta que auxilie na determinação da complexidade, visto que a maior parte das implementações de algoritmos são desenvolvidas sem a preocupação de se documentar sua complexidade. Além disso, é conveniente lembrar que a situação atual da área de software é tal que uma parte considerável dos programadores não possuem formação específica na área de computação (ou, mais especificamente, não foram educados para conduzirem análises assintóticas), e por isso muitos programas reais utilizados pela indústria carecem da informação associada quanto à complexidade das centenas ou milhares de funções sendo executadas. A possibilidade de se verificar a complexidade de funções embutidas em tais programas por parte dos clientes, via abordagem empírica é, evidentemente, desejável.

Marco teórico

O estudo de complexidade de algoritmos é considerado de grande importância desde a criação dos modelos de computação, isto é, já havia a preocupação dos estudiosos da época quanto ao tempo de computação de um algoritmo.

> "As soon as analytic engine exists, it will necessarily guide the future course of science. Whenever any result is sought by its aid, the question will raise – By what means of calculation can these results be arrived at by this machine in the shortest time?" (BABBAGE, 1864)

> "It's convenient to have a measure of the amount of work involved in a computing process, even if it has to be a very crude one. We may count up the number of things that various times at various elementary operations are applied in the whole process." (TURING, 1947)

Diversos pesquisadores contemporâneos dedicam seus estudos à análise de complexidade de algoritmos para resolução de problemas. Entre tais pesquisadores, um dos mais importantes é Donald E. Knuth, autor da famosa série de livros *The Art of Computer Programming* [2]. Knuth enfatiza a importância da análise de algoritmos no prefácio do livro de outro pesquisador, Robert Sedgewick [3], da área de algoritmos, em que diz

> "Mathematical models have been a crucial inspiration for all scientific activity, even though they are only approximate idealizations of real-world phenomena. Inside a computer, such models are more relevant than ever before, because computer programs create artificial worlds in which mathematical models often apply precisely. I think that's why I got hooked on analysis of algorithms when I was a graduate student, and why the subject has been my main life's work ever since."

Knuth compara a área de análise de algoritmos com outras áreas científicas, onde os modelos matemáticos são propostos e é verificada a aderência do modelo à realidade via experimentação. Esta observação reforça a importância também da experimentação na área de análise de algoritmos, como defendida por Sedgewick em seu curso *Analysis of Algorithms* no Coursera [4].

Definições

São descritas nesta seção definições e notações que serão encontradas em diversas seções e capítulos deste trabalho. Entretanto, definições e notações utilizadas apenas em capítulos ou seções específicas serão descritas naquele escopo. Para notações omitidas no trabalho, refira-se a [5, 6].

Análise de Algoritmos

Um *algoritmo* é uma sequência finita de instruções, com cada instrução bem definida, que leva à solução de um problema. Um algoritmo resolve um problema a partir de informações inicialmente conhecidas, chamadas de *dados de entrada* (ou, simplesmente, *entrada*), e produzem informações que representam a solução do problema, chamadas de *dados de saída* (ou, simplesmente, *saída*). Abstratamente, um algoritmo é a implementação concreta de uma função matemática, que mapeia o conjunto de entradas possíveis (domínio) no conjunto das respectivas saídas (contra-domínio).

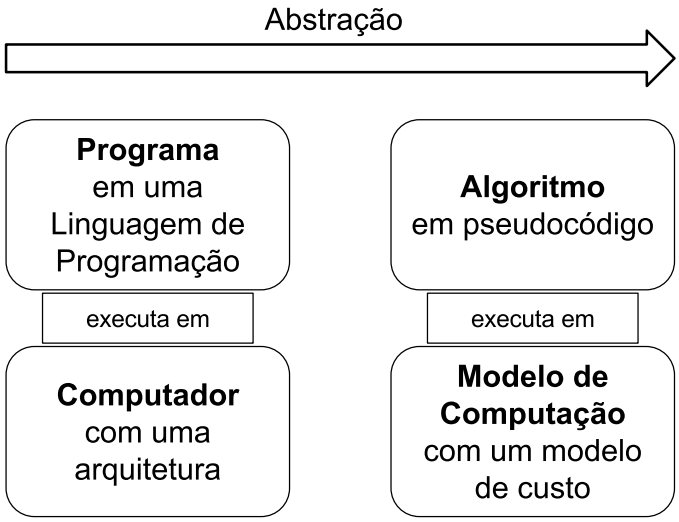
O objetivo da *análise de algoritmos* é mensurar a quantidade de recursos necessários pelo algoritmo quando executados. O motivo principal para analisar um algoritmo é prover um meio de comparar algoritmos que resolvem o mesmo problema por estratégias distintas. Os recursos de interesse são usualmente tempo (número de instruções a serem executadas até o término) e espaço (quantidade de células de memória necessárias para armazenar a entrada, os cálculos intermediários e a saída). É importante atentar-se à estratégia da implementação, propriedades dos compiladores, arquiteturas das máquinas, entre outras particularidades que podem causar grandes efeitos na complexidade do algoritmo. Deve-se ter consciência desses efeitos para garantir que os resultados da análise são úteis [3].

A análise visa compreender um algoritmo de um ponto de vista distinto da sua preocupação principal, que é a da correção (as saídas produzidas devem estar corretas em relação às entradas). A análise procura compreender a quantidade dos recursos computacionais necessária para sua execução. Algoritmos simples do ponto de vista de correção

podem ter análise difícil de ser conduzida e algoritmos de correção complexa podem ser facilmente analisados no que diz respeito aos recursos consumidos.

Queremos que o processo de contagem de recursos seja o mais abstrato possível, de modo que possa ser aplicado a uma maior quantidade de computadores. Como se pode imaginar, não existe um único modelo de computação, pois as diferenças de engenharia dos computadores podem ser tais que impactam substancialmente a contagem de recursos sendo utilizados. Por isso, em um processo de análise de algoritmos, é preciso especificar um modelo da tecnologia e dos recursos de implementação que será usada, chamado de *modelo de computação* e de como se dá a contagem de recursos, chamado de *modelo de custo*. A Figura 1 exemplifica modelo de computação e modelo de custo, através da abstração de um programa implementado em uma linguagem de programação que executa em computador com dada arquitetura.

Figura 1 – Modelo de computação e modelo de custo.



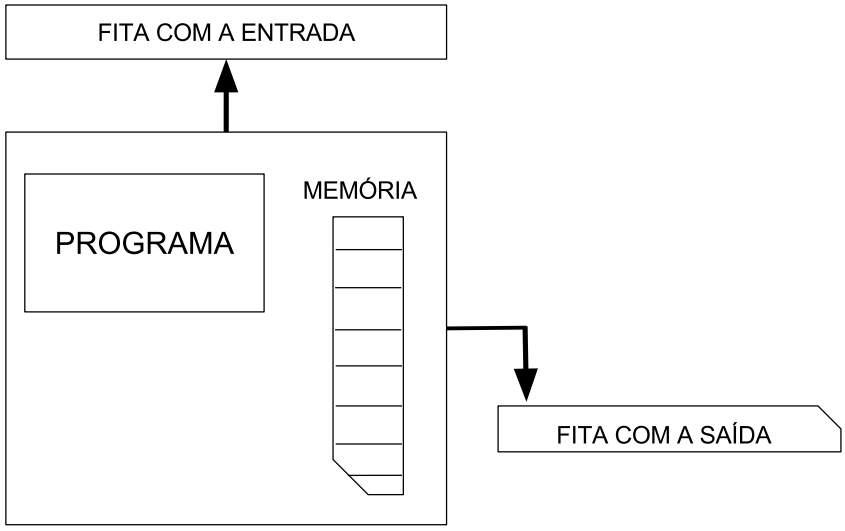
O *modelo de computação RAM* (random-access machine - máquina de acesso aleatório) é utilizado neste trabalho por modelar bem a arquitetura dos computadores pessoais e a natureza dos algoritmos que serão analisados. Este modelo supõe a existência de um único processador, nas quais instruções são executadas uma após outra, sem operações paralelas. Além disso, possui instruções encontradas em computadores reais, nas quais tempos devem ser considerados constantes, como por exemplo instruções aritméticas, movimentação de dados, desvios no fluxo de execução, chamadas e retornos de funções. O modelo RAM não leva em conta se um dado está em cache ou no disco, o que simplifica a análise do algoritmo. A Figura 2 ilustra o modelo de computação RAM, onde o tamanho da entrada é igual ao número de símbolos na fita de entrada. Cada célula de memória é uma unidade de espaço, nas quais valores são guardados e retornados para serem usados em operações. O programa executa instruções, tais como adicionar, subtrair, armazenar valor em célula, resgatar valor de célula, entre outros.

O modelo de custo pode ser (i) *unitário ou uniforme*, na qual o tempo de uma operação envolvendo um número constante de células leva um passo ou (ii) *logarítmico*, na qual o tempo de uma instrução envolvendo k bits leva k passos [7]. O modelo de custo utilizado neste trabalho, se nada for dito a tal respeito, é o *modelo unitário*.

Seja $T(N)$ a função que representa a quantidade utilizada de um certo recurso por um

algoritmo para um problema com uma entrada de tamanho N; tal função é chamada de *complexidade do recurso* em relação a este algoritmo. Para valores pequenos de N, tal algoritmo terá consumo desprezível do recurso, dado que $T(N)$ provavelmente será baixo. Portanto, o interesse é compreender o comportamento do algoritmo diante de valores grandes de N. Em função disso, estuda-se o comportamento assintótico de $T(N)$, onde procura-se obter uma aproximação do valor de $T(N)$ para N suficientemente grande.

Figura 2 – Modelo de computação RAM.



Seja A um algoritmo, $E(N)$ o conjunto de todas as entradas possíveis de tamanho N para A. Seja $R(E)$ a quantidade de recursos utilizada por A quando a entrada for E. Define-se a complexidade de:

- *pior caso* como a função $T_{\max}(N) = \max\{R(E) : E \in E(N)\}$;
- *melhor caso* como a função $T_{\min}(N) = \min\{R(E) : E \in E(N)\}$;
- *caso médio* como a função $T_{\text{med}}(N) = \sum\{p(E)R(E) : E \in E(N)\}$, onde $p(E)$ é a probabilidade de ocorrência da entrada E.

A complexidade de pior caso é a mais usual de ser considerada das três, pois fornece um limite superior para a quantidade de recursos que o algoritmo pode efetuar em qualquer caso, e por esse motivo, é muito utilizada na prática. Quando nenhuma menção ao tipo de complexidade estiver presente no contexto, assume-se que se trata desta complexidade.

A análise de algoritmos tipicamente não tem como objetivo a determinação da função propriamente dita que relaciona uma entrada específica a quantidade de recursos utilizada pelo algoritmo em questão sob aquela entrada. Isto se deve não somente à dificuldade em estabelecê-la em geral, mas também ao fato de que, para fins de comparação de algoritmos, podemos nos preocupar com um objetivo menor, mais fácil de ser obtido e ainda útil para se comparar algoritmos. Tal objetivo é aquele de determinar nesta função, seja ela qual for, apenas o seu termo de maior crescimento (esta análise é chamada de *análise assintótica*). Esta abordagem despreza todas as constantes aditivas, multiplicativas e todos os termos, exceto aquele de maior crescimento assintótico da função em questão. As técnicas de análise de algoritmos usualmente conduzem a análise assintótica e, via

notação O e família, descrevem o comportamento assintótico da função que representa o tempo de execução sob o tamanho de uma determinada entrada. Sejam f e g funções com domínio e contradomínio no conjunto dos naturais e tal que $\lim_{n\to\infty} f(n)/g(n)$ exista. As principais notações assintóticas utilizadas ao longo deste trabalho são descritas a seguir.

Notação O

Dizemos que $f(n) = O(g(n))$ quando $n \to \infty$ se existem números reais positivos c e n_0 tais que $f(n) \leq cg(n)$, para todo $n \geq n_0$. Alternativamente, dizemos que $f(n) = O(g(n))$ se

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

A notação $O(g(n))$ também é usada para denotar o conjunto de todas as funções $f(n)$ que satisfazem a relação $f(n) = O(g(n))$. Como exemplo, suponha $f(n) = 5n^4 + n^3 + 2$. Aplicando-se a definição formal, a afirmação de que $f(n) = O(n^4)$ é equivalente a afirmar que $f(n) \leq cg(n)$ para todo $n \geq n_0$, para certas constantes c, n_0. De fato, seja $n_0 = 1$ e $c = 8$. Então, para todo $n > n_0$,

$$\begin{aligned} 5n^4 + n^3 + 2 &\leq 5n^4 + n^4 + 2n^4 \\ &= 8n^4. \end{aligned}$$

Informalmente, a notação O é utilizada para denotar um limite superior assintótico da função. Por exemplo, seja $f(n) = 3n^3 + 9n^2 + 1$ e deseja-se simplificar esta função usando a notação O. O termo que possui maior taxa de crescimento é o que possui maior expoente em função de n, ou seja, $3n^3$. Eliminando a constante multiplicativa, temos que $f(n) = O(n^3)$.

Notação Ω

De forma análoga à notação O, que fornece um limite assintótico superior de uma função, a notação Ω fornece um limite assintótico inferior. Dizemos que $f(n) = \Omega(g(n))$ se existem constantes positivas c e n_0 tais que $0 \leq cg(n) \leq f(n)$ para todo $n \geq n_0$. Por exemplo, $n^2 \log_2 n = \Omega(n^2)$ (como se pode verificar para $c = 1$ e $n_0 = 2$) mas $n \log_2 n$ não é $\Omega(n^2)$. Alternativamente, dizemos que $f(n) = \Omega(g(n))$ se

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0.$$

Notação o

Analogamente à notação O que denota um limite superior que pode ser ou não assintoticamente justo, a notação o é usada para denotar um limite superior que não é assintoticamente justo. Por exemplo, o limite $3n^3 = O(n^3)$ é assintoticamente justo ao passo que $3n^2 = O(n^3)$ não é.

Dizemos que $f(n) = o(g(n))$ se para qualquer constante $c > 0$ existe uma constante $n_0 > 0$ tal que $0 \leq f(n) < cg(n)$ para todo $n \geq n_0$. Por exemplo, $3n = o(n^2)$ (como se pode verificar para $c = 1$ e $n_0 = 4$) mas $3n^2$ não é $o(n^2)$. Vale ressaltar que a notação o é mais forte que a notação O de maneira que toda função que é $o(g(n))$ também é $O(g(n))$, porém, nem toda função que é $O(g(n))$ também é $o(g(n))$. Equivalentemente, podemos dizer que $f(n) = o(g(n))$ se

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Notação ω

A notação ω é usada para denotar um limite inferior que não é assintoticamente justo. Dizemos que $f(n) = \omega(g(n))$ se para qualquer constante $c > 0$ existe uma constante $n_0 > 0$ tal que $0 \leq cg(n) < f(n)$ para todo $n \geq n_0$. Por exemplo, $n^3 + 2n^2 + n = \omega(n^2)$ (como se pode verificar para $c = 1$ e $n_0 = 1$) mas $100n^2 + 5n$ não é $\omega(n^2)$. Alternativamente, dizemos que $f(n) = \omega(g(n))$ se

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

Notação Θ

Dizemos que $f(n) = \Theta(g(n))$ se existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0$. Como exemplo, vamos mostrar que $5n^2 - 100n \log_2 n + 10 = \Theta(n^2)$. Isto é, escolheremos constantes positivas c_1, c_2 e n_0 tais que

$$c_1 n^2 \leq 5n^2 - 100n \log_2 n + 10 \leq c_2 n^2$$

para todo $n \geq n_0$. Dividindo-se por n^2, temos

$$c_1 \leq 5 - \frac{100 \log_2 n}{n} + \frac{10}{n^2} \leq c_2.$$

Escolhendo-se $c_1 = 1$, a desigualdade da esquerda é válida para todo $n \geq 200$. Para o lado direito da desigualdade, resolvendo a inequação para $n \geq 200$, é possível notar que com o aumento de n a expressão do meio e da direita (onde aparece n) tornam-se cada vez menores em valor absoluto, logo, a desigualdade da direita é válida para $c_2 = 5$. Portanto, para $c_1 = 1$, $c_2 = 5$ e $n_0 = 200$, verifica-se que $5n^2 - 100n \log_2 n + 10 = \Theta(n^2)$. Vale ressaltar que existem outros valores possíveis para tais contantes. A notação Θ também pode ser definida como $f(n) = \Theta(g(n))$ se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Informalmente, a notação Θ é utilizada para denotar o termo de maior crescimento omitindo as constantes multiplicativas. Por exemplo, para $f(n) = 5n$, é dito que a função é $\Theta(n)$, ou ainda, que $f(n) = \Theta(n)$. Outro exemplo é a função $g(n) = 6n^3 + 7n + 5$, que pode ser denotada por $g(n) = \Theta(n^3)$.

Notação $\sim$

Dizemos que $f(n) \sim g(n)$ (f é assintoticamente igual a g), se para todo $\epsilon > 0$ existe n_0 tal que $\frac{f(n)}{g(n)} - 1 < \epsilon$ para todo $n \geq n_0$. Alternativamente dizemos que $f(n) \sim g(n)$ se

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

Informalmente, a notação *tilde* é usada para obter expressões aproximadas mais simples nas quais apenas os termos de menor crescimento da função são ignorados [8]. Difere da notação Θ apenas pelo fato da constante multiplicativa do termo de maior crescimento não ser omitida. Por exemplo,

$$\frac{n(n+1)}{2} \sim \frac{n^2}{2}; \quad 2n + 10 \sim 2n; \quad 3n^3 + 2n^2 + 8 \sim 3n^3.$$

Em análise assintótica, um algoritmo é dito ser *preferível* a outro quando sua função que descreve o consumo de determinado recurso é assintoticamente menor que a do outro.

Por exemplo, se o algoritmo A executa com complexidade de $\Theta(N)$ e o algoritmo B executa com aquela de $\Theta(N \log N)$, o algoritmo A é preferível.

As complexidades podem ser expressas também em função de variáveis que representam grandezas envolvidas na entrada, não necessariamente o tamanho, em função de uma ou mais variáveis. Enquanto funções de uma única variável N possam ser expressas sob a notação $\Theta(h(N))$ onde $h(N)$ consiste de um único termo, funções de duas ou mais variáveis podem necessitar de um número arbitrário de termos em sua expressão assintótica [9].

Existem duas abordagens para analisar a complexidade de um algoritmo. A primeira é o método analítico, baseado na contagem de passos apenas por inspeção da descrição do algoritmo e raciocínio sobre o fluxo de execução para diferentes entradas. O Algoritmo 1 é um exemplo de como a abordagem analítica pode determinar a complexidade de tempo.

Algoritmo 1 BubbleSort. Complexidade de Tempo: $\Theta(N^2)$.

```
1: procedimento ORDENAR(B[ ], N)
2:     para i ← N até 1 passo −1 faça
3:         para j ← 1 até i − 1 faça
4:             se B[j] > B[j + 1] então
5:                 t ← B[j]
6:                 B[j] ← B[j + 1]
7:                 B[j + 1] ← t
```

São contados o número de execuções de cada linha do Algoritmo 1 para então inferir sua complexidade de pior caso, de acordo com a Tabela 1.

Tabela 1 – Número de execuções do Algoritmo 1.

Linha	**Nº de execuções**
2	$= N + 1$
3	$= N(N + 1)/2$
4	$= N(N - 1)/2$
5	$\leq N(N - 1)/2$
6	$\leq N(N - 1)/2$
7	$\leq N(N - 1)/2$

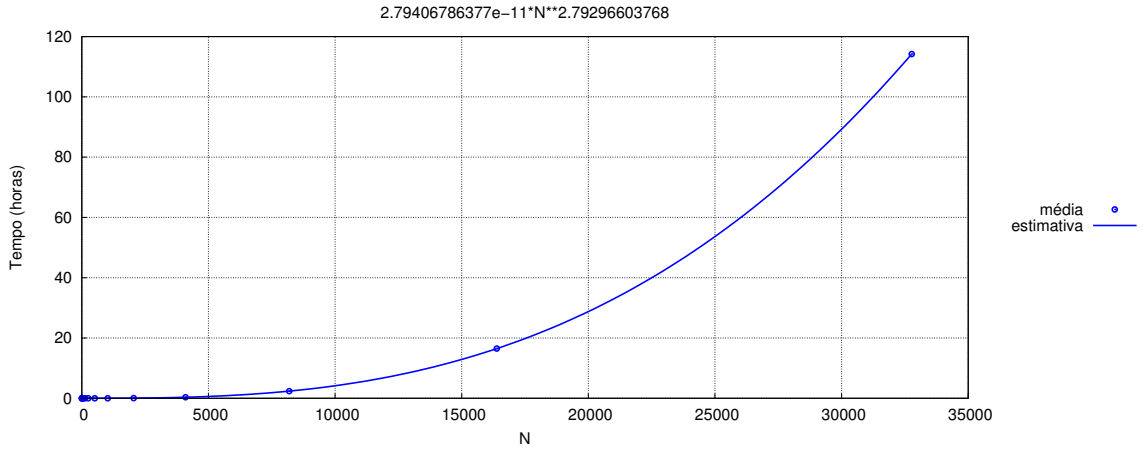
A soma de todas as execuções da Tabela 1, resulta em $\frac{5}{2}N^2 - (\frac{1}{2})N + 1$ número de passos no pior caso, ou seja, $\Theta(N^2)$. Contudo, nem sempre é possível inferir a complexidade de um algoritmo através da abordagem analítica. Em geral, é um problema indecidível, pois analisar a complexidade implica em decidir se o dado algoritmo pára para dada entrada [10]. Por exemplo, não se sabe se o Algoritmo 2, conhecido como *problema de Collatz*, pára para todo valor de N, embora este seja o caso para todo $N \leq 5.7646 \times 10^{18}$ [11].

Algoritmo 2 Conjectura de Collatz. Complexidade de Tempo: Em aberto.

```
1: enquanto N > 1 faça
2:     se N é par então
3:         N ← N/2
4:     senão
5:         N ← 3N + 1
```

A segunda abordagem é o método empírico, na qual são realizadas medições do tempo de execução do algoritmo sobre diferentes conjuntos de entradas, sendo necessária a implementação do algoritmo em alguma linguagem de programação. Os resultados deste método sofrem influências de outras variáveis, como do compilador e da arquitetura de máquina em que está sendo executado. A Figura 3 exemplifica graficamente a abordagem empírica.

Figura 3 – Gráfico tempo vs. tamanho de entrada, resultante de uma análise empírica de um algoritmo.



Proposta

A proposta desta pesquisa é comparar ferramentas que fazem análise automatizada de algoritmos, além de documentar detalhadamente a ferramenta EMA, incluindo sua metodologia [12, 13]. Essa comparação será feita através da realização de experimentos com diversos tipos de algoritmos e estruturas de dados comparando-se qualitativamente os resultados produzidos por tais ferramentas. Também é proposta uma metodologia chamada *teste do Big-Enough* que visa obter o menor tamanho de entrada para o qual verifica-se empiricamente a complexidade analítica e aplicá-la ao EMA para algoritmos selecionados.

Estrutura

O restante do trabalho está assim organizado. O Capítulo 1 apresenta um resumo das principais ferramentas para análise automatizada de algoritmos existentes na literatura. No Capítulo 2, apresenta-se a metodologia e as funcionalidades da ferramenta EMA. O Capítulo 3 descreve a análise automatizada de alguns algoritmos através de ferramentas a fim de compará-las qualitativamente. O Capítulo 4 descreve uma metodologia de análise empírica chamada *teste do Big Enough* que visa determinar o valor inicial de tamanho de entrada para o qual verifica-se empiricamente a complexidade analítica. Por fim, são apresentadas as conclusões e os trabalhos futuros.

1 ANÁLISE AUTOMATIZADA DE ALGORITMOS

A análise de complexidade é componente fundamental do estudo de algoritmos. Embora a pesquisa de ferramentas que produzem a complexidade assintótica de algoritmos via método empírico não seja recente, há poucas ferramentas disponíveis. Neste capítulo, apresentaremos as metodologias e ferramentas encontradas na literatura.

Ferramentas que fazem análise automatizada de algoritmos, em particular, através do método empírico, podem fornecer uma maneira mais fácil de avaliar a complexidade de um algoritmo e podem ser úteis em muitos cenários. O uso didático, de forma a verificar se o uso de recursos do algoritmo está de acordo com sua complexidade teórica, é um exemplo, dado que uma má escolha de estrutura de dados pode elevar a complexidade do algoritmo [14–16]. Além disso, essas ferramentas podem medir empiricamente a complexidade de tempo e espaço de sistemas nos quais diversos algoritmos são empregados, como aplicações de terceiros nais quais não se tem acesso ao código-fonte ou a complexidade é desconhecida analiticamente. Um exemplo é o uso de SGBDs [17] em sistemas de informação.

Problemas podem ser resolvidos através de diversos algoritmos que possuem a mesma complexidade assintótica. Neste caso, ferramentas que fornecem a complexidade de algoritmos empiricamente podem ajudar a escolher qual o melhor na prática, analisando qual algoritmo possui menor constante multiplicativa oculta na notação assintótica [18]. Existem aplicações em que o espaço é um parâmetro tão importante quanto o tempo, de maneira que um algoritmo que minimiza apenas o tempo de execução, ignorando a sobrecarga de espaço, pode ser impraticável. Portanto, pode ser relevante prever a quantidade de tempo/memória necessária para execução do algoritmo para uma entrada desconhecida [19].

Apesar da complexidade de pior caso ser a mais popular e mais estudada, esta análise tem o defeito de ocultar o real comportamento de alguns algoritmos, que processam a vasta maioria das entradas em tempo pequeno e em apenas algumas entradas pouco prováveis em tempo maior. Neste caso, a análise mais útil é a de caso médio. Um exemplo é o QuickSort, cujo tempo de pior caso é $\Theta(n^2)$. Apesar de tal pior caso, na prática, o QuickSort possui notável eficiência, o que se explica pela sua conhecida complexidade de caso médio $\Theta(n \log n)$, nas quais os termos constantes ocultos nesta notação são pequenos [16, 20, 21]. Em casos como o do Quicksort, a análise via abordagem analítica em geral é muito difícil, sendo problema em aberto para diversos algoritmos, como é o caso de variações do ShellSort [22] e a análise da árvore de busca SplayTree [23].

Quando usa-se extensivamente memória auxiliar, a comparação analítica entre algoritmos pode ser difícil [24]. Em razão dessas situações, uma ferramenta capaz de fornecer a complexidade de tempo e espaço de algoritmos empiricamente é de muita importância.

Em 1975, Wegbreit desenvolveu o METRIC [25], escrito em Interlisp, que foi a primeira aplicação automatizada com o objetivo de obter as complexidades de melhor, pior e caso médio de algoritmos. Sua metodologia consiste em ler programas escritos na linguagem de programação LISP e analisar a complexidade segundo as medidas de desempenho apresentadas por Knuth [2]. Durante sua execução são contabilizados empiricamente os

passos básicos do algoritmo que são convertidos em equações de recorrência a fim de resolvê-las e finalmente obter a complexidade do programa em questão. No entanto, o METRIC limita-se a tratar apenas programas simples, tais como retornar ou substituir um elemento de uma lista, inverter e unir listas. O METRIC foi importante pois, além de inovador, introduziu a ideia de automatizar a análise de complexidade de algoritmos e motivou diversas pesquisas anos depois, como apresentadas a seguir.

Ao final da década de 80 surgiu o Automatic Complexity Evaluator (ACE) [26], desenvolvido por Le Métayer. Seu objetivo é a análise de pior caso de algoritmos implementados em linguagens funcionais utilizando uma base de dados com regras que transformam as especificações de programas em equações de complexidade. O ACE faz a análise do algoritmo a partir de uma função de complexidade de tempo derivada de um programa funcional inicial e a converte em uma função não-recursiva equivalente através de uma biblioteca predefinida de definições recursivas. Ao terminar a conversão dessas equações, a expressão de complexidade será uma composição de funções já conhecidas, sendo elas: exponenciais, polilogarítmicas e polinomiais.

No mesmo ano, Flajolet, Salvy e Zimmermann desenvolveram o Lambda-Upsilon-Omega ($\lambda\Upsilon\Omega$) [27], sistema de análise automática de caso médio de classes restritas de procedimentos puramente funcionais em termos de funções geradoras. O $\lambda\Upsilon\Omega$ é dividido em dois subprogramas: o Sistema Analisador Algébrico (ALAS) e o Sistema Analisador Análitico (ANANAS). O ALAS, escrito em CAML, que é uma extensão da linguagem ML, decompõe os algoritmos em estruturas de dados que em seguida também são decompostas para gerar equações. O ANANAS recebe essas equações para realizar o cálculo final da complexidade através de uma coleção de funções algébricas escritas em Maple e a forma assintótica é extraída dos coeficientes das funções geradoras.

Ao final da década de 90, foi desenvolvido por Morelli o Analisador de Complexidade Média (ACME) [28], para análise de complexidade de caso médio de um algoritmo escrito em estilo Pascal de maneira análoga a de um compilador. Sua metodologia possui analisadores léxico e sintático a fim de desenvolver um cálculo baseado nas estruturas contidas no código-fonte e gerar uma equação de complexidade não simplificada e portanto, complexa de ser compreendida pelo usuário. Por ser um sistema que avalia apenas a semântica dos comandos de um programa, o ACME não fornece a complexidade de algoritmos recursivos.

Em 2001, motivado pelas ferramentas já descritas, Barbosa desenvolveu um sistema chamado ANAC [29]. Esta ferramenta assemelha-se mais ao ACME por implementar análise do código e trabalhar com algoritmos não-recursivos escritos em estilo Pascal ao passo que o METRIC faz análise empírica de programas escritos em LISP. O ANAC auxilia o cálculo de complexidade de algoritmos não-recursivos no pior caso ou no caso médio, contendo as principais estruturas presentes em linguagens imperativas: atribuição, sequência, condicional, iteração e condicional. Esta ferramenta determina de maneira semi-automática a equação de complexidade e a resolve quando os dados necessários são fornecidos para essa operação. Caso o usuário defina procedimentos no sistema, é necessário que ele forneça os dados essenciais para que o sistema reconheça o procedimento e calcule sua complexidade. O ANAC faz primeiramente uma análise sintática do algoritmo e obtém sua função de complexidade e em seguida simplifica essa função, se possível, pois esta ferramenta faz uma simplificação de funções que se resume apenas a casos mais simples e triviais.

Em 2007, Goldsmith, Aiken e Wilkerson desenvolveram uma ferramenta chamada *Trend Profiler* [30] que visa descrever o comportamento assintótico de programas em C na prática medindo sua complexidade computacional empírica. Este método prevê quantas

vezes cada bloco básico de um programa é executado como uma função do tamanho da entrada. O desempenho do programa é representado através do modelo linear ($y = a+bx$) ou lei de potência ($y = ax^b$) e seus parâmetros determinados por regressões lineares.

Burnim, Juvekar e Sen [31] desenvolveram uma metodologia em 2009 chamada WISE (acrônimo de Worst-case Inputs from Symbolic Execution). Este é um algoritmo de teste de complexidade com propósito de fazer uma geração automática de testes para encontrar erros de desempenho. O WISE executa em um programa que aceita entradas de tamanhos variados, na qual, para cada tamanho de entrada, o algoritmo tenta construir uma entrada que retorne a complexidade computacional do pior caso do programa que está sendo analisado. O WISE utiliza uma geração de testes exaustiva para tamanhos de entrada pequenos e generaliza o resultado da execução do programa nessas entradas em um "gerador de entrada", que é usado depois para gerar de maneira eficiente as entradas de pior caso para entradas de tamanho maiores. Os resultados experimentais encontrados pelos autores demonstram que a técnica de teste de complexidade WISE pode efetivamente encontrar grandes entradas de pior caso para muitos algoritmos e operações de estrutura de dados, porém não é claro se o WISE pode ser usado em aplicações maiores com vários componentes. Além disso, a técnica não foi implementada em uma linguagem de programação.

Em 2012 Coppa, Demetrescu e Finocchi criaram uma ferramenta baseada no *Valgrind* [32] chamada *Aprof* [33] com o objetivo de ajudar desenvolvedores a descobrirem ineficiências assintóticas ocultas no código. O Aprof mede o desempenho de rotinas individuais em função do tamanho da entrada a partir de uma ou mais execuções do programa. Além disso, retorna para cada rotina executada do programa um conjunto de tuplas que contém os custos de desempenho por tamanho de entrada, fornecendo seus gráficos de desempenho. Esta ferramenta também mede automaticamente o tamanho da entrada de uma rotina genérica contando o número de células de memória distintas acessadas pela rotina com uma operação de leitura.

No mesmo ano surgiu o *AlgoProf* [34], ferramenta com o objetivo de determinar automaticamente as entradas de programas em Java, medir seu custos para qualquer entrada e inferir uma função de custo empírico aproximada, ou seja, uma função custo real esperada e não o pior caso ou um caso idealizado. O AlgoProf é baseado em heurísticas que servem para identificar fronteiras entre diferentes algoritmos em um programa. Além disso, só pode inferir uma função de custo para algoritmos que utilizam estruturas de dados, porque não pode inferir qualquer tamanho de entrada para algoritmos que operam em tipos de dados primitivos. Para fornecer as funções de custo, esta metodologia não utiliza regressões e nem heurísticas, porém, não foi descrito pelos autores como a função é estimada.

Entre os estudos mais recentes sobre ferramentas que realizam análise automática de algoritmos está o MOCCA (Measurement of Complexity of a Certain Algorithm) [35], desenvolvido em 2014 por um grupo de pesquisadores da Universidade Federal da Paraíba. Este sistema propõe um modelo de classificação de complexidade automática que realiza classificações assintóticas a partir de uma função custo $T(n)$ que representa a medida de tempo necessária para executar um algoritmo para um problema de tamanho n. A metodologia do MOCCA consiste em um Avaliador de Eficiência que gera diversas instâncias aleatórias de tamanhos n como entrada para o algoritmo e armazena o número de comandos contabilizados pelo contador de comandos da ferramenta. Esses valores são guardados em uma Tabela de Eficiência, utilizada para gerar a função de custo do processamento $T(n)$ de um algoritmo e então plotar o gráfico de desempenho. Além disso, sua classificação assintótica de eficiência pode ser gerada através de um método matemático

chamado Interpolação Polinomial de Lagrange [36]. O MOCCA foi implementado em Python e classifica algoritmos implementados também em Python.

Atualmente, existe um ambiente chamado RAML (Resource Aware ML) [37] capaz de fazer análise de pior caso de um algoritmo em tempo de compilação através do conceito de análise amortizada multivariada [38] para calcular limites de recursos polinomiais para programas funcionais de primeira ordem. São utilizadas duas máquinas métricas, nas quais uma é para avaliação da semântica e a outra para medição de espaço de heap. Finalmente, obtém-se a classificação do algoritmo através da função de complexidade. A análise de recursos amortizados multivariados foi implementada em RAML, uma linguagem funcional que possui sintaxe semelhante a ML. A ferramenta é fácil de usar e pode ser executada diretamente da Internet via navegador. O RAML faz análise de complexidade apenas de programas recursivos escritos na linguagem funcional OCaml e só fornece complexidades polinomiais.

Além do RAML, dentre os trabalhos mais atuais há uma ferramenta chamada EMA [1], principal objeto de estudo deste trabalho. O EMA foi criado em 2015 pelo orientador desta pesquisa e atualmente continua em desenvolvimento. O EMA fornece a função de complexidade assintótica a partir de dados obtidos durante a execução de algoritmos escritos em qualquer linguagem de programação e inclusive fornece os produtos das análises empíricas usuais, como gráficos de tempos vs. tamanho da entrada. Sua metodologia consiste basicamente em executar diversas simulações com a finalidade de obter um conjunto de entradas ideal para em seguida fazer diversas execuções do algoritmo em questão para o conjunto de entradas obtido anteriormente. São executadas várias regressões não-lineares a fim de determinar os parâmetros de funções que descrevem o desempenho do algoritmo. A partir de um conjunto de funções, uma é selecionada como a que melhor representa a complexidade do algoritmo. A metodologia do EMA será descrita no Capítulo 2.

A Tabela 2 resume as ferramentas para análise automatizada de algoritmos encontradas na literatura já descritas neste capítulo. É possível observar que a tentativa de desenvolver ferramentas com este objetivo ocorre desde a década de 70 através do METRIC, que foi pioneira neste assunto e motivou as diversas ferramentas que surgiram nos anos seguintes. Ferramentas como ACE, $\lambda\Upsilon\Omega$, ACME, ANAC e RAML fornecem a complexidade de algoritmos através da análise do código-fonte, ao passo que o METRIC, Trend Profiler, Aprof, AlgoProf, MOCCA e EMA são ferramentas que analisam um algoritmo a partir de sua execução. Além disso, é possível observar que a maior parte dessas ferramentas possuem muitas limitações, tais como análise de algoritmos implementados em um determinado paradigma de programação e em uma linguagem de programação específica. Além disso, apenas as ferramentas Trend Profiler, Aprof, AlgoProf, RAML e EMA estão disponibilizadas para uso. Algumas dessas ferramentas fornecem apenas a complexidade de pior caso, enquanto outras fornecem a complexidade de caso médio, ou ambas. Dentre todas as ferramentas apresentadas, apenas o RAML e o EMA estão em desenvolvimento ativo. O EMA analisa algoritmos escritos em qualquer linguagem de programação, enquanto o RAML apenas algoritmos escritos na linguagem OCaml.

Tabela 2 – Ferramentas para análise automatizada de algoritmos.

Ferramenta	**Ano**	**Complexidade**	**Programa de Entrada**	**Abordagem**	**Desenvolvimento Ativo**	**Disponível**
METRIC [25]	1975	Melhor, pior e caso médio	LISP	Empírica	Não	Não
ACE [26]	1988	Pior caso	Linguagens funcionais	Analítica	Não	Não
$\lambda\Upsilon\Omega$ [27]	1988	Caso médio	Proprietário	Analítica	Não	Não
ACME [28]	1998	Caso médio	Estilo Pascal	Analítica	Não	Não
ANAC [29]	2001	Pior caso e caso médio	Estilo Pascal	Analítica	Não	Não
Trend Profiler [30]	2007	Melhor, pior e caso médio	C	Empírica	Não	Sim
Aprof [33]	2012	Melhor, pior e caso médio	C	Empírica	Não	Sim
AlgoProf [34]	2012	Função custo real esperada	Java	Empírica	Não	Sim
MOCCA [35]	2014	Pior caso e caso médio	Python	Empírica	Não	Não
RAML [37]	2012	Pior caso	OCaml	Analítica	Sim	Sim
EMA [1]	2015	Melhor, pior e caso médio	Qualquer	Empírica	Sim	Sim

2 ANÁLISE AUTOMATIZADA PELO EMA

O EMA (acrônimo de *EMpirical Analysis of algorithms*) [1] é uma ferramenta com o objetivo de fazer análise empírica de um algoritmo de forma automática. Na análise conduzida pelo EMA, o algoritmo é executado para uma quantidade de entradas de tamanhos variados e avaliam-se os recursos consumidos a cada execução para então inferir funções que representam o consumo de diversos recursos em função de variáveis de entrada. A qualidade da estimação é influenciada pelas restrições dadas pelo o usuário das quais os experimentos são dependentes, tais como tempo/espaço máximo permitido de execução, número de amostras de cada tamanho de entrada, número distinto de tamanhos de entrada, entre muitos outros. O EMA é a ferramenta a ser estudada neste capítulo, e será comparada posteriormente com outras ferramentas. Neste capítulo são descritas as funcionalidades e a metodologia do EMA. As principais funções contidas da biblioteca do EMA estão descritas no Apêndice B para detalhes operacionais, de modo que interessados em executar o EMA em seus próprios algoritmos possam usá-lo como referência.

2.1 Execução do EMA

O EMA faz a análise empírica do algoritmo monitorando o tempo de execução e memória consumida no processo executado, inclusive em todos os subprocessos gerados.

A entrada do EMA consiste em três dados básicos:

(i) um programa $\mathcal{A}$ a ser analisado (por exemplo, $\mathcal{A}$ pode ser um programa de ordenação, como o InsertionSort);

(ii) uma lista $V = v_1, v_2, ..., v_N$ de variáveis das quais a complexidade de tempo/espaço de $\mathcal{A}$ depende (no exemplo do InsertionSort, sabe-se que seu tempo de execução e memória alocada depende do número N de elementos da lista a ser ordenada e, portanto, V consiste de uma única variável $v_1 = N$);

(iii) um programa $\mathcal{B}$ que gera entradas para o programa $\mathcal{A}$. Mais especificamente, este programa recebe como entrada uma valoração desejada para as variáveis em V e $\mathcal{B}$ produz uma entrada para $\mathcal{A}$ com tal valoração das variáveis (no exemplo do InsertionSort, $\mathcal{B}$ deve ser um programa que tem por entrada um valor específico de N e, como saída, gera um vetor com N números inteiros; tais números podem ser aleatórios para que a complexidade de caso médio seja analisada ou podem estar ordenados descendentemente para que a complexidade de pior caso seja analisada).

O processamento completo do EMA consiste em três etapas: calibração, simulação e análise. Contudo, nem todas as etapas são obrigatórias. As Seções 2.3, 2.4 e 2.5, deste capítulo descrevem mais especificamente cada uma delas. A Seção 2.2 detalha o método de regressão utilizado na fase de análise do EMA para estimação de parâmetros de funções que mais se ajustam às medições.

2.2 Método de regressão

Nesta seção é feita uma breve descrição dos conceitos de regressão que serão necessários para o entendimento da metodologia do EMA. Veremos adiante que o EMA utiliza a implementação do algoritmo de Levenberg–Marquardt [39, 40] encontrada na ferramenta Gnuplot [41] em sua fase de análise.

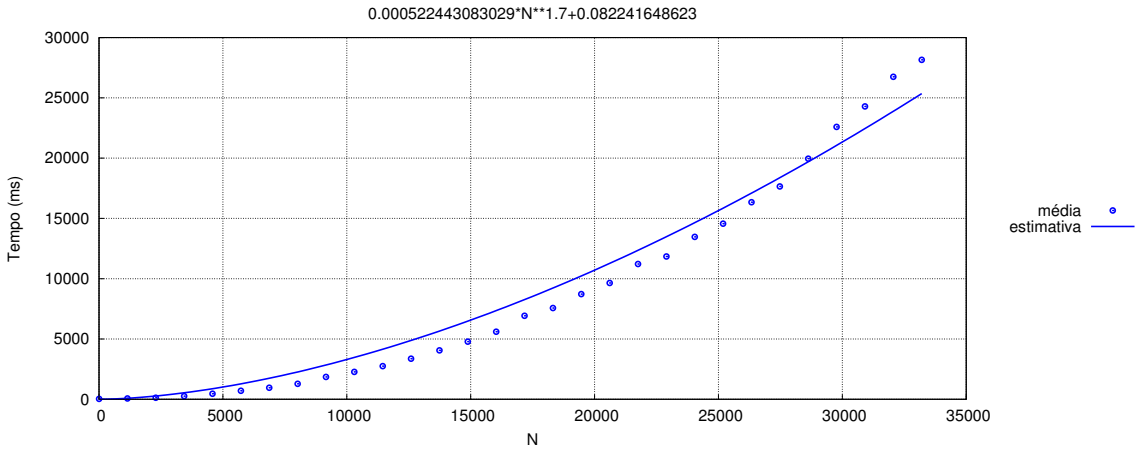
Seja D um conjunto de pontos $\{(x_i, y_i) : 1 \leq i \leq N\}$, tal que para todo $1 \leq i \leq N$, temos que $x_i \in \mathbb{N}, y_i \in \mathbb{N}$. Considere o problema de obter uma função $f : \mathbb{N} \to \mathbb{N}$ que melhor se ajusta a D, isto é, que $f(x_i)$ seja o mais próximo possível de y_i, para todo $1 \leq i \leq N$. Para tanto, precisamos definir uma métrica de erro associada a cada função candidata f, e o método de escolha deve visar minimizar o erro. Tal método para obter a função que melhor se ajusta ao conjunto D é chamado de *regressão*. Mais especificamente, seja f a função que relaciona uma variável dependente com uma ou mais variáveis independentes, que pode ser representada por

$$Y = f(X),$$

onde Y representa a variável dependente e valores de X representam as variáveis independentes. A regressão é dita *linear* quando f representa a equação de uma reta e *não-linear* caso contrário.

Para proceder a análise de como se comportam as variáveis X e Y, podemos plotar um gráfico chamado de *diagrama de dispersão* onde os valores de D são colocados sob as coordenadas cartesianas como mostrado na Figura 4 e então verificar qual função dentre várias pertencentes a diferentes classes de um modelo matemático mais se aproxima dos pontos representados no diagrama de dispersão. A função que melhor se ajusta aos pontos de D pode vir da classe das funções lineares, quadráticas, cúbicas, funções exponenciais, logarítmicas, dentre outras possíveis [42]. Para o exemplo da Figura 4, é possível verificar que os pontos do diagrama de dispersão, que significam digamos a média de tempos de execução obtidos como resultado em uma série de experimentos para cada valor da variável N, não se ajustam perfeitamente a curva traçada (estimativa), pois há uma distância entre os pontos do diagrama e a curva do modelo matemático. Talvez outra função, da mesma classe de função ou de classe distinta, possa se adequar mais aos pontos do gráfico.

Figura 4 – Exemplo de gráfico de dispersão e uma função estimada.



Um dos principais métodos utilizados para obter tal ajuste chama-se *Método dos Mínimos Quadrados* (MMQ). O MMQ é uma técnica de otimização matemática que busca o

melhor ajuste para um conjunto de pontos minimizando o valor de $erro(f)$, que representa a medida do erro de estimativa de D associado à uma função f, sobre todas as funções candidatas sendo consideradas, através da soma dos quadrados dos erros residuais em cada ponto, definido como

$$erro(f) = \sum_{i=1}^{N} e_i^2(f), \tag{2.1}$$

onde $e_i(f)$ é o *erro residual* em cada ponto $(x_i, y_i) \in D$ definido por

$$e_i(f) = f(x_i) - y_i. \tag{2.2}$$

Algoritmo de Levenberg–Marquardt

O *Algoritmo de Levenberg–Marquardt* (LMA) [39, 40], também conhecido como *método dos mínimos quadrados amortecidos*, é um método de otimização usado para resolver problemas de mínimos quadrados não-lineares. Assim como a maioria dos algoritmos utilizados para este fim, o LMA procura apenas um mínimo local, que não é necessariamente o mínimo global. Apresentaremos no Apêndice A uma descrição desta metodologia conforme aquela encontrada em [43].

2.3 Calibração

O objetivo da etapa de calibração consiste do EMA sugerir uma lista de valores de cada variável em V para as quais o programa $\mathcal{A}$ deverá executar. No exemplo do InsertionSort, isto significaria determinar uma lista com os números de elementos para os quais o algoritmo será executado. Esta fase não é obrigatória para o EMA pois pode ser determinada de maneira absoluta (*hard-coded*) ou gerada de acordo com um algoritmo elaborado pelo usuário. Mas de forma geral, é desejável o uso desta fase, de forma a deixar o processo de execução completamente automatizado.

O processo de calibração é descrito da seguinte forma. Primeiro, determina-se os maiores valores possíveis $\mathrm{max}_1, \ldots, \mathrm{max}_\mathrm{N}$ que satisfazem a seguinte propriedade: para toda entrada I na qual $\mathrm{v_i} \leq \mathrm{max_i}$, para todo $1 \leq i \leq N$, o tempo de execução e o espaço de memória utilizado por $\mathcal{A}$ são limitados a valores determinados pelo usuário. Em outras palavras, tais valores garantem que qualquer entrada com valoração de variáveis menor que os respectivos máximos atenderá os limites máximo de tempo e espaço desejados. No exemplo do InsertionSort, se max_1 fosse determinado ser, digamos, 40 290, isto significaria que vetores com este número de elementos satisfazem os limites de recursos impostos, mas estão perto de não atendê-los. Portanto, todas as execuções a serem feitas devem ser com vetores de tamanho menores ou iguais a 40 290.

Os valores $\mathrm{max}_1, \ldots, \mathrm{max}_\mathrm{N}$ são descobertos através do seguinte processo. A partir de um valor mínimo estabelecido pelo usuário para cada variável, o EMA executa o programa $\mathcal{A}$ e monitora tempo e memória desta execução. Para tanto, o EMA utiliza o programa $\mathcal{B}$ para gerar uma entrada para $\mathcal{A}$ do valor desejado para cada variável $\mathrm{v_i}$. Tal valoração das variáveis cresce exponencialmente até atingir o limite de tempo ou memória. No momento que descobre uma valoração que viola algum limite de execução, o EMA passa a guardar dois valores para cada variável: o primeiro, indica qual foi o maior valor para o qual se sabe não violar os limites de execução; o segundo, é a menor valoração para os quais sabe-se violar tais limites. O EMA procura o limiar entre não-violação e violação destes limites através de pesquisa binária. Mais especificamente, o EMA escolhe uma valoração

intermediária para cada variável no intervalo sendo mantido de cada uma, e realiza a execução. Dependendo do resultado (violação ou não), o EMA elimina uma metade do intervalo (segunda ou primeira, respectivamente).

Essa fase de calibração é executada através de uma função da biblioteca do EMA chamada *getSuggestedVariableValues*, que retorna uma lista $\mathrm{l} = [\mathrm{lst}_1, \mathrm{lst}_2, \ldots, \mathrm{lst}_\mathrm{N}]$ de valores de variáveis sugeridas para uso na fase de simulação (que será descrita mais adiante), onde cada valor lst_i é uma lista de valores a serem usados para a variável v_i. A lista lst_i é produzida de tal forma que o maior valor é max_i e os demais valores são todos distribuídos uniformemente.

Por exemplo, supondo que deseja-se calcular a complexidade de um programa que multiplica duas matrizes de tamanho $N \times N$. Se for usado o algoritmo de multiplicação que decorre da definição, V consiste de uma única variável N. Como resultado da calibração é retornado uma lista $\mathrm{l} = [\mathrm{lst}_1]$, onde $\mathrm{lst}_1 = [\mathrm{N}_1, \mathrm{N}_2, \ldots, \mathrm{N}_\mathrm{k}]$, representando que o programa $\mathcal{A}$ deverá ser executado para cada $N \in \{\mathrm{N}_1, \ldots, \mathrm{N}_\mathrm{k}\}$.

2.4 Simulações

De maneira geral, a fase de simulação consiste em executar o programa a ser analisado para os diversos valores de variáveis determinados na fase de calibração (ou, de maneira mais geral, em uma lista de valores associada a cada variável, que foram geradas automaticamente pela calibração ou pelo usuário). Cada execução é feita com um valor específico de cada variável dentre os que foram selecionados. Durante cada execução, o EMA deve monitorar e, ao término, guardar em um arquivo o consumo de recursos associado a tal valoração das variáveis.

Nesta fase, é especificado o número de amostras de cada valor de variável. Realizar a experimentação com várias amostras é importante quando o algoritmo de geração de entradas $\mathcal{B}$ gera uma entrada não-determinística, isto é, realiza sorteios para a criação da entrada (por exemplo, se $\mathcal{A}$ é o InsertionSort, $\mathcal{B}$ pode usar a estratégia de escolher aleatoriamente os elementos do vetor; como o tempo de execução pode variar significativamente com o sorteio realizado, um número de amostras apropriado permite lidar com o problema do sorteio de entradas atípicas). O número de amostras pode ser fixo ou ser determinado durante a execução por um fator de convergência da média amostral. O EMA armazena como estatística para cada recurso os valores máximo, mínimo, a média amostral e o desvio padrão da amostra para cada valor de variável simulada.

Mais formalmente, a fase de simulação tem por entrada um subconjunto $S_i \subseteq \{1, \ldots, \mathrm{max}_\mathrm{i}\}$ para todo $1 \leq i \leq N$ com valores de v_i que serão analisados. Para cada $(V_1, \ldots, V_N) \in S_1 \times \ldots \times S_N$, o algoritmo $\mathcal{B}$ é usado para gerar uma entrada I de $\mathcal{A}$ na qual $v_i = V_i$ para todo $1 \leq i \leq N$, e, em seguida, $\mathcal{A}$ é executado tendo I por entrada. O EMA monitora a execução e armazena em sua base de dados as estatísticas de consumo de cada recurso. Portanto, a base de dados é gerada com $\prod_{i=1}^{N} |S_i|$ execuções. A simulação é iniciada através da chamada à função *runSimulation* de acordo com os parâmetros definidos pelo usuário, conforme Apêndice B.

2.5 Análise

Apesar das etapas de calibração e simulação poderem ser feitas com múltiplas variáveis, atualmente a fase de análise do EMA pode ser feita apenas com uma variável. A seguir, descrevemos tal processo de análise.

Dado um conjunto de pontos $D = \{(x_i, y_i) : 1 \leq i \leq n\}$ onde para cada $1 \leq i \leq n$, x_i é o valor de uma variável associada a uma instância de entrada em particular e y_i é o tempo, memória ou outro recurso despendido na execução de tal entrada, o objetivo é obter a função f que melhor se ajusta a D. É necessário formalizar o conceito de "melhor" aqui. Em primeiro lugar, não se trata necessariamente de encontrar uma função f tal que $f(x_i) = y_i$, para todo $1 \leq i \leq n$. Caso contrário, sabemos que existe (exatamente) um polinômio de grau $n - 1$ que satisfaz esta propriedade, que pode ser obtido através da regra de Crammer, da seguinte forma. Consideramos que

$$f(x) = \sum_{i=0}^{n-1} a_i x^i$$

e, para cada $1 \leq i \leq n$, a substituição $f(x_i) = y_i$ produz um sistema de n equações e n variáveis, que pode então ser resolvido. No entanto, esta abordagem é inadequada no mínimo por duas razões:

(i) a complexidade do algoritmo não depende do número de pontos; e

(ii) como alguns recursos (por exemplo, tempo) podem variar ligeiramente entre várias execuções com a mesma entrada, e não podemos aceitar que se conclua diferentes funções a partir de execuções distintas com variações na medição dentro do considerado normal.

Outra distinção que é importante ter em mente que o objetivo do EMA não é o de encontrar a função real que representa o consumo de recursos com todos os seus monômios, mas a função assintótica que a representa. Isto é em geral suficiente para a análise de algoritmos e tornará substancialmente mais simples a tarefa, tanto quanto torna mais simples a obtenção da complexidade via método analítico.

Portanto, a expressão "melhor função" deve ter por objetivo encontrar uma função que comumente representa o comportamento assintótico de funções relacionadas a consumo de recursos computacionais. A estas funções, Knuth chamou de *logarithmico-exponential* [44]. Como veremos a seguir, o EMA considerará então um conjunto bem-definido de tipos de função para investigar o quão bem podem ser ajustadas no conjunto de pontos, uma métrica de erro para qualificar uma função, o conceito de funções equivalentes (conjunto de funções cujos erros estão mutuamente muito próximos), e por fim um critério para escolher uma função equivalente como a mais provável que representa o consumo de recurso.

Para fazer os ajustes dos parâmetros da função, o EMA aplica regressões não-lineares utilizando o algoritmo de Levenberg–Marquardt, que está detalhado no Apêndice B. A seguir, trataremos de descrever mais especificamente o processo da análise particular ao EMA, extra ao processo de regressão, incluindo exemplos. A metodologia geral do EMA está expressa através das Figuras 5, 6 ou, alternativamente, do Algoritmo 3, e será detalhadamente explicada nas próximas seções.

Algoritmo 3 Metodologia do EMA.

1: seja $f_{\text{geral}}(x) = a_0 \underbrace{a_1^{x^{a_2}}}_{t_1} \underbrace{x^{a_3}}_{t_2} \underbrace{(\log_2 x)^{a_4}}_{t_3} + a_5$
2: seja $\mathcal{F} = \{f : f$ é obtida de uma combinação de termos t_1, t_2, t_3 de $f_{\text{geral}}\}$
3: $\mathcal{F}_{\text{ajustadas}} \leftarrow \varnothing$
4: **enquanto** $\mathcal{F} \neq \varnothing$ **faça**
5: seja $f \in \mathcal{F}$
6: $\mathcal{F} \leftarrow \mathcal{F} \setminus \{f\}$
7: $g_{\min} \leftarrow \varnothing$
8: **para cada** $g \in \{f, f_{\log}\}$ **faça**
9: seja $par(g)$ os parâmetros de g
▷ para cobrir diversas ordens de magnitude dos parâmetros iniciais, conforme Seção 2.5.1
10: **para cada** valor inicial *valini* de $par(g)$ **faça**
11: *convergido* ← FALSO
12: **repita**
▷ através do LMA, conforme Seção 2.2
13: *valpar* ← AJUSTAR-POR-REGRESSÃO(g, *valini*)
14: $g_{\min} \leftarrow \text{argmin}\ \{\ erro(z) : z \in \{g_{\min}, g\}\ \}$
▷ para cada $a \in par(g)$, $valpar(a)$ é o valor de a retornado da regressão
15: **se** existe x,$y \in valpar$ tal que $x/y < 10^{-1}$ ou $x/y > 10$ **então**
16: $g \leftarrow$ SUBSTITUIR(g, "a", "$(valpar(a) * a)$"), para todo $a \in par(g)$
17: $valini(a) \leftarrow 1$, para todo $a \in par(g)$
18: **senão**
19: *convergido* ← VERDADEIRO
20: **até** *convergido*
21: $\mathcal{F}_{\text{ajustadas}} \leftarrow \mathcal{F}_{\text{ajustadas}} \cup \{g_{\min}\}$
22: **se** existe valor não-inteiro em $valpar(g_{\min})$ **então**
23: seja $a \in par(g_{\min})$ tal que $valpar(a)$ é não-inteiro
24: $\mathcal{F} \leftarrow \mathcal{F} \cup arr(g_{\min}, a)$
▷ Classificação das funções
25: $f_{\min} \leftarrow \text{argmin}\ \{\ erro(z) : z \in \mathcal{F}_{\text{ajustadas}}\}$ ▷ função de erro mínimo
▷ funções viáveis e inviáveis, conforme Seção 2.5.3
26: $F_{\text{viaveis}}, F_{\text{inviaveis}} \leftarrow$ PARTICIONAR-POR-VIABILIDADE($\mathcal{F}_{\text{ajustadas}}$)
▷ funções equivalentes
27: $F_{\text{equiv}} \leftarrow \{f \in \mathcal{F}_{\text{ajustadas}} : erro(f) \leq erro(f_{\max})\}$, onde $f_{\max}$ é definida em (2.4)
▷ função melhor-palpite, conforme Seção 2.5.3
28: $F_{\text{melhor-palpite}} \leftarrow$ DETERMINAR-MELHOR-PALPITE(F_{equiv})
29: **reportar** $(f_{\min}, F_{\text{viaveis}}, F_{\text{inviaveis}}, F_{\text{equiv}}, F_{\text{melhor-palpite}})$

Figura 5 – Metodologia da etapa de análise do EMA.

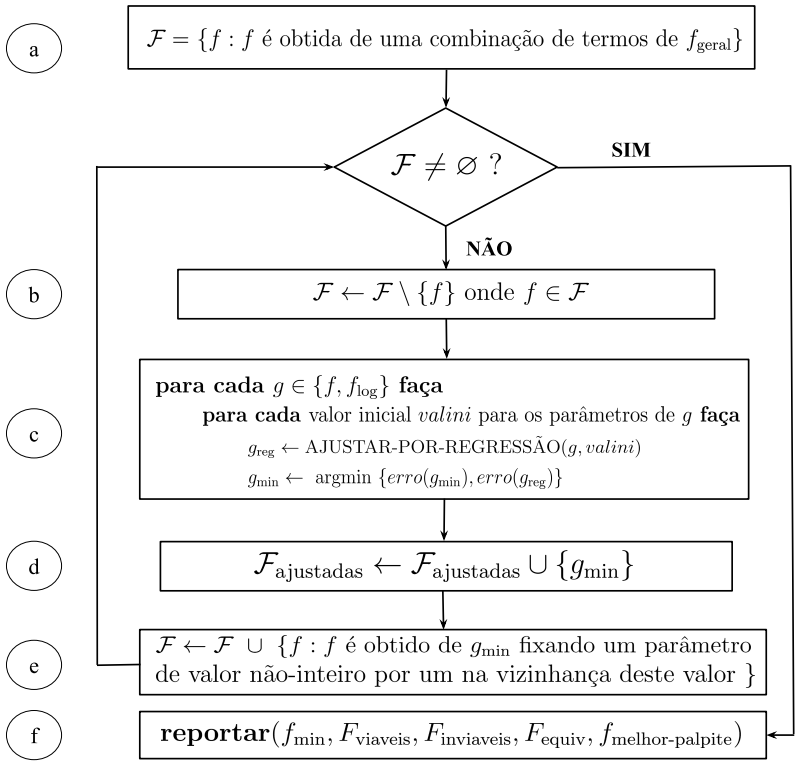
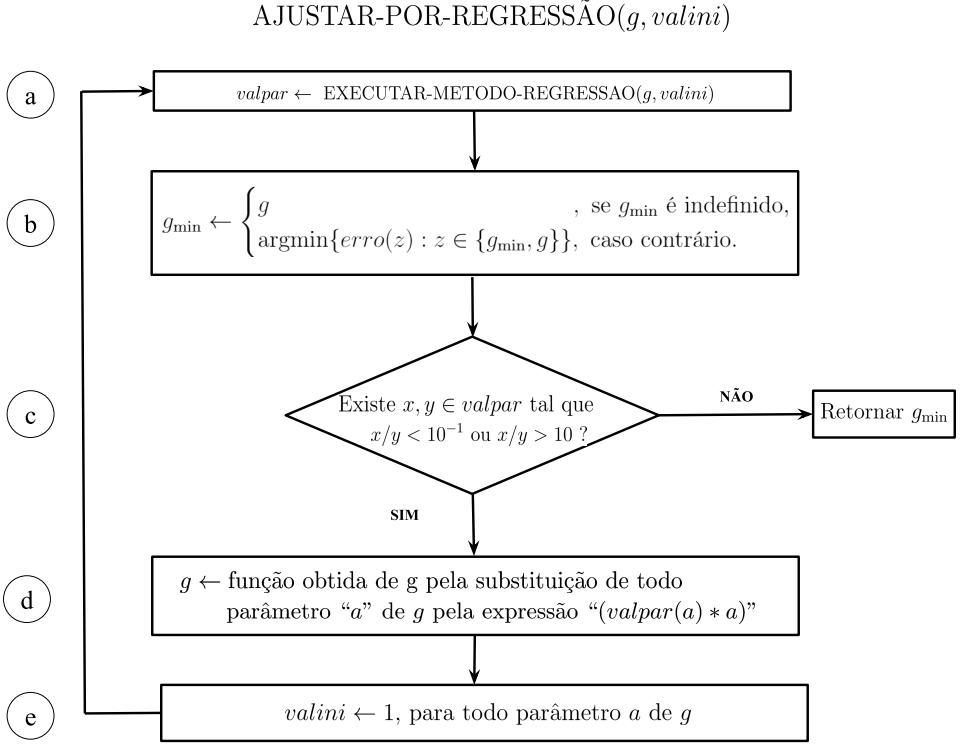


Figura 6 – Metodologia do ajuste por regressão da etapa de análise do EMA.



2.5.1 Determinação dos parâmetros

Seja $f_{\text{geral}}(x)$ a função mais geral que o EMA supõe descrever a complexidade de um algoritmo, cuja expressão é dada por

$$f_{\text{geral}}(x) = a_0 \underbrace{a_1^{x^{a_2}}}_{t_1} \underbrace{x^{a_3}}_{t_2} \underbrace{(\log_2 x)^{a_4}}_{t_3} + a_5, \tag{2.3}$$

onde $a_0 \geq 10^{-7}, a_2, a_3, a_4,\ a_5 \geq 0$ e $a_1 \geq 1$ são os *parâmetros* determinados pelo EMA e t_1,t_2,t_3 são os *termos* da expressão. O parâmetro a_1 é chamado de *parâmetro-base*, os parâmetros a_2, a_3, a_4 de *parâmetros-expoentes*, o parâmetro a_0 de *parâmetro-multiplicativo* e a_5 de *parâmetro-aditivo*. Portanto, o EMA tem por hipótese que a função que representa o consumo de recursos possui como forma geral aquela de f_{geral} e que os parâmetros atendem as restrições descritas. Qualquer restrição considerada nos valores dos parâmetros que não elimina a função real é importante para diminuir o espaço de busca e, por consequência, diminuir o tempo de análise.

Para determinar os valores dos parâmetros e quais deles são relevantes para definir a função, a estratégia do EMA é obter funções mais particulares formadas por todas as combinações possíveis entre os termos da função (2.3) e testá-las individualmente, a fim de encontrar a expressão que melhor define o conjunto de pontos encontrados através de funções mais específicas (Figura 5(a)). Em outras palavras, inicialmente, define-se o conjunto $\mathcal{F}$ de funções candidatas como sendo

$$\mathcal{F} = \{f(x) : f(x) \text{ é obtido de } f_{\text{geral}}(x) \text{ pela remoção de um subconjunto dos termos}\}.$$

Logo, $\mathcal{F}$ é composto inicialmente pelas funções:

1. $f(x) = a_0 + a_5$
2. $f(x) = a_0 x^{a_3} + a_5$
3. $f(x) = a_0 (\log_2 x)^{a_4} + a_5$
4. $f(x) = a_0 a_1^{x^{a_2}} + a_5$
5. $f(x) = a_0 x^{a_3} (\log_2 x)^{a_4} + a_5$
6. $f(x) = a_0 a_1^{x^{a_2}} x^{a_3} + a_5$
7. $f(x) = a_0 a_1^{x^{a_2}} (\log_2 x)^{a_4} + a_5$
8. $f(x) = a_0 a_1^{x^{a_2}} x^{a_3} (\log_2 x)^{a_4} + a_5$

Além destas, são colocadas também em $\mathcal{F}$ as funções abaixo, obtidas das funções 4,6,7,8 com o parâmetro a_1 fixado no valor 2, devido a base 2 ser muito comum em expressões que determinam complexidade de algoritmos.

9. $f(x) = a_0 2^{x^{a_2}} + a_5$
10. $f(x) = a_0 2^{x^{a_2}} x^{a_3} + a_5$
11. $f(x) = a_0 2^{x^{a_2}} (\log_2 x)^{a_4} + a_5$
12. $f(x) = a_0 2^{x^{a_2}} x^{a_3} (\log_2 x)^{a_4} + a_5$.

A base do termo polilogarítmico é sempre fixada em 2, pois uma base pode ser obtida da outra alterando-se a_0 e, portanto, são equivalentes do ponto de vista de ajuste.

Para determinar os parâmetros de cada $f \in \mathcal{F}$ (Figura 5(b)) são executadas regressões não-lineares através do algoritmo de Levenberg-Marquardt que visa minimizar o erro associado a cada função. No Apêndice B é explicado em detalhes o método numérico da regressão usada pelo EMA. O erro de uma função é computado através da expressão (2.1).

Para exemplificar, considere a função $f(x) = a_0 x^{2.5} (\log_2 x)^3 + a_5$, e vejamos o quão bem esta função específica representa o conjunto de pontos da Tabela 3, que descrevem o tempo de execução y_i em função do tamanho x_i de certo aspecto da entrada (ou seja, de uma variável). Foram calculados os erros residuais para cada ponto de acordo com (2.2).

Tabela 3 – Conjunto de pontos e seus respectivos erros residuais.

i	x_i	y_i(ms)	$f(x)$	$e_i(f)$	$e_i^2(f)$
1	2	6	1.00001057028	4.99998942972	24.9998942973
2	4	6	1.00065118201	4.99934881799	24.993488604
3	8	10	1.00842775286	8.99157224714	80.8483714756
4	16	10	1.0861412857	8.9138587143	79.4568771784
5	32	10	1.78388997019	8.21611002981	67.5044640219
6	64	35	7.63547803967	27.3645219603	748.817062117
7	128	103	54.4382821757	48.5617178243	2358.24043805
8	256	540	415.955688022	124.044311978	15386.9913342
9	512	3600	3134.72157068	465.278429321	216484.016791
10	1024	24770	23152.6522269	1617.3477731	2615813.81914
11	2048	174246	168031.326831	6214.67316924	38622162.6004
12	4096	1211009	1201773.19503	9235.80497438	85300093.5249
13	8192	8475612	8489979.81075	14367.8107514	206433985.788
14	16384	59355401	59351926.5677	3474.43225538	12071679.4972
15	32768	411183596	411183831.029	235.028955102	55238.6097363

O erro da função é calculado através de (2.1). Logo, $erro(f) = 345334229.708$.

O EMA utiliza um conjunto de técnicas para auxiliar a regressão que, de maneira padrão, não resulta em geral na função real. Por conta de instabilidades numéricas que podem afetar o processo de ajuste, o EMA realiza diversas tentativas de ajuste, ficando com a que obteve o melhor resultado (Figura 5(c)). A estimação dos parâmetros de cada função é feita em sua forma original ($y = f(x)$) e em escala logarítmica ($\log y = \log f(x)$). Esta segunda forma é particularmente útil quando os valores máximos de x,y atingem valores extremos próximos ao limite de representação computacional, quando há potencial chance de estouro deste limite durante os cálculos intermediários do método numérico. Por exemplo, suponha $y = a_0 x^{a_3} + a_5$. Logo,

$$\begin{aligned} y_{\log} \quad = \quad \log y \quad &= \log\left(a_0 x^{a_3} + a_5\right) \\ &\xrightarrow{x\to\infty} \log a_0 + a_3 \log x \\ &= a_3 x_{\log} + \log a_0 = f_{\log}(x_{\log}). \end{aligned}$$

A análise é conduzida então usando a função original $y = f(x)$ e a função na escala logarítmica $y_{\log} = f_{\log}(x_{\log})$. A vantagem de ajustar na escala logarítmica é que a função

original transforma-se em uma função linear. Assim, o método de regressão tem menos dificuldades em encontrar os parâmetros por ser menos suscetível a instabilidades numéricas provocadas por tentativas do método de regressão de avaliar a função para valores grandes de parâmetros na base de uma exponencial ou em um expoente, estourando as representações numéricas.

Como visto na seção sobre a regressão, é necessário informar valores iniciais para os parâmetros. É sabido que se tais valores iniciais estiverem em ordens de grandeza afastados dos valores ótimos globais dos parâmetros, a qualidade de ajuste pode ser prejudicada pois pode-se convergir para ótimos locais. Portanto, os métodos numéricos necessitam de valores iniciais relativamente próximos aos ótimos globais. Devido a isso, o EMA então testa um conjunto de valores iniciais de ordens de magnitude diferentes para o parâmetro a_0. Especificamente, a_0 inicial é testado para cada valor no conjunto $\{10^x : x \in \mathbb{Z} : -4 \leq x \leq 4\}$. O valor inicial dos parâmetros-expoentes é 1, do parâmetro-base é 2 e do parâmetro a_5 é 1 assumindo-se que as tais parâmetros nas funções reais estão na vizinhança de tais valores. Como exemplo, suponha $f(x) = a_0 x^{a_3}(\log_2 x)^{a_4} + a_5$. Os valores iniciais de a_0 são testados para cada valor do conjunto $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 10^2, 10^3, 10^4\}$, enquanto os demais parâmetros possuem como valor inicial $a_3 = 1$, $a_4 = 1$ e $a_5 = 1$.

O método de regressão é sensível se os valores ótimos dos parâmetros são de ordens de magnitude diferentes, fazendo com que ele possa convergir prematuramente. Como exemplo, suponha que o ajuste de parâmetros será conduzido para a função $f(x) = a_0 x^{a_3}(\log_2 x)^{a_4} + a_5$ que representa a complexidade de tempo. Note que, dada a natureza do que está sendo representado, é esperado que os parâmetros a_0 e a_3 sejam de ordens de magnitude diferentes (em geral, $10^{-6} \leq a_0 \leq 10^{-2}$ e $0 \leq a_3 \leq 10$). Assim, tal ajuste pode enfrentar dificuldades na execução do método numérico. O EMA adota a seguinte proteção. Após um ajuste de parâmetros (Figura 6(a)), para cada valor ajustado v do parâmetro a (Figura 6(c)), é testada uma nova função, refinamento do ajuste encontrado, fazendo-se uma troca de variáveis. O EMA faz uma nova regressão de modo que os valores ajustados sejam os valores iniciais, mas ao invés de a é usado em seu lugar a expressão va. Ou seja, para cada valor ajustado v do parâmetro a (Figura 6(d)), experimenta-se uma nova função com o parâmetro a substituído por "(va)" com o valor inicial $a = 1$ (Figura 6(e)). Retornando ao exemplo anterior, suponha que a regressão convergiu para $v_0 = 1.4 \times 10^{-5}$, $v_3 = 1.98$, $v_4 = 1.01$ e $v_5 = 5.6 \times 10^4$, respectivamente valores de a_0, a_3, a_4, a_5. Como os valores encontrados estão em ordens de magnitude distintas, o EMA ajustará agora a função, derivada da anterior, $f(x) = (v_0 a_0) x^{(v_3 a_3)}(\log_2 x)^{(v_4 a_4)} + (v_5 a_5)$, com valores iniciais para a_0, a_3, a_4, a_5 iguais a 1, de modo que o método numérico nesta nova regressão comece exatamente do mesmo ponto em que parou na regressão anterior, porém com as variáveis agora na mesma ordem de magnitude. Para cada ajuste encontrado, é feita a troca de parâmetros para que todos tenham a mesma ordem de magnitude. Este processo se repete até que $10^{-1} \leq v \leq 10$ para todo valor v de parâmetros (Figura 6(c)). Ao final, o EMA armazena em um conjunto $\mathcal{F}_{\text{ajustadas}}$ a função ajustada $g_{\min}$ de menor erro encontrada como o ajuste final de f (Figura 5(d) e 6(b)).

2.5.2 Discretização dos parâmetros

Os parâmetros das equações são encontrados através de um método numérico, portanto, frequentemente os valores encontrados são não-inteiros. Por outro lado, é muito comum que as funções reais que medem recursos computacionais possuam como valores dos parâmetros determinados números especiais, como os inteiros. Por esse motivo, o próximo passo do EMA é discretizar os parâmetros, operação que consiste de testar funções

obtidas de cada função que teve ao menos um parâmetro convergido para um valor não-inteiro v através da substituição de v para valores especiais na vizinhança de v, conforme será detalhado a seguir.

Seja v valor de um parâmetro qualquer a não-aditivo e não-multiplicativo de uma função f. O conjunto de funções de *arrendondamento* de f com respeito a a, denotado por $arr(f,a)$, é o conjunto de funções obtidas a partir de f trocando-se a por cada valor no conjunto $\{arr(v), arr(v) \pm 0.5, arr(v) \pm 1\}$, onde $arr(v)$ é o valor de $\{0.5x : x \in \mathbb{N}\}$ mais próximo de v (se há dois valores mais próximos, desempata-se pelo menor). Assim, $arr(1.25) = 1, arr(1.3) = 1.5, arr(1.1) = 1$.

Para cada função f de $\mathcal{F}$ com parâmetros ajustados, é incluído em $\mathcal{F}$ as funções de $arr(f, a)$ para algum parâmetro-base ou expoente cujo valor não esteja no conjunto $\{0.5x : x \in \mathbb{N}\}$, para posterior análise (Figura 5(e)).

Para exemplificar esse passo de discretização, suponha que a função e os parâmetros encontrados pela regressão foram, respectivamente, $f(x) = a_0x^{a_1} + a_5$, com $a_0 = 0.1362$, $a_1 = 2.79$ e $a_5 = 1.002$. Com respeito a a_1, temos que $arr(2.79) = 3$ e $arr(f, a_1) = \{f_1, f_2, f_3, f_4, f_5\}$, onde $f_1(x) = a_0x^3 + a_5$, $f_2(x) = a_0x^{2.5} + a_5$, $f_3(x) = a_0x^2 + a_5$, $f_4(x) = a_0x^{3.5} + a_5$ e $f_5(x) = a_0x^4 + a_5$. Eventualmente, cada uma dessas funções serão objetos da regressão a fim de encontrar os valores dos parâmetros a_0 e a_5.

A partir de cada uma das 12 funções iniciais contidas em $\mathcal{F}$, são geradas a partir delas, através do processo de regressão e discretização, novas funções com diferentes valores de parâmetros. Portanto, ao final, $\mathcal{F}$ possuirá um número bem maior de funções comparado com aquele das iniciais. Isto ocorre pois o arredondamento de uma função f com respeito a um parâmetro é incluído em $\mathcal{F}$, e isto ocorre recursivamente para cada nova função.

Como ilustração, suponha $f(x) = a_0x^{a_1}(\log_2 x)^{a_4} + a_5$ e que $a_1 = 2.12$ e $a_4 = 1.98$ sejam os parâmetros encontrados pela regressão. Primeiramente, é feita a discretização de f com respeito à a_1 e em seguida à a_4 conforme ilustra a Figura 7. Os primeiros filhos da raiz correspondem a discretização do parâmetro a_1. Uma nova regressão é aplicada para determinar os valores do parâmetro a_4, que também são discretizados e representados nas folhas da árvore.

Por fim, são feitas novas regressões para encontrar os valores dos parâmetros a_0 e a_5 para cada função. Estes valores estão omitidos na Figura 7 porque não são considerados pelo processo de discretização. É possível notar neste exemplo que, além da função com os parâmetros não-discretizados, com respeito à a_1 foram geradas 5 funções e com respeito ao parâmetro a_4 foram geradas outras 25 funções, totalizando 31 funções apenas para $f(x) = a_0x^{a_1}(\log_2 x)^{a_4} + a_5$.

Este processo de encontrar e discretizar parâmetros é repetido para cada $f \in \mathcal{F}$. Ao final, o EMA possui uma lista com diversas funções candidatas a representar a complexidade do algoritmo ordenadas crescentemente pelo erro associado.

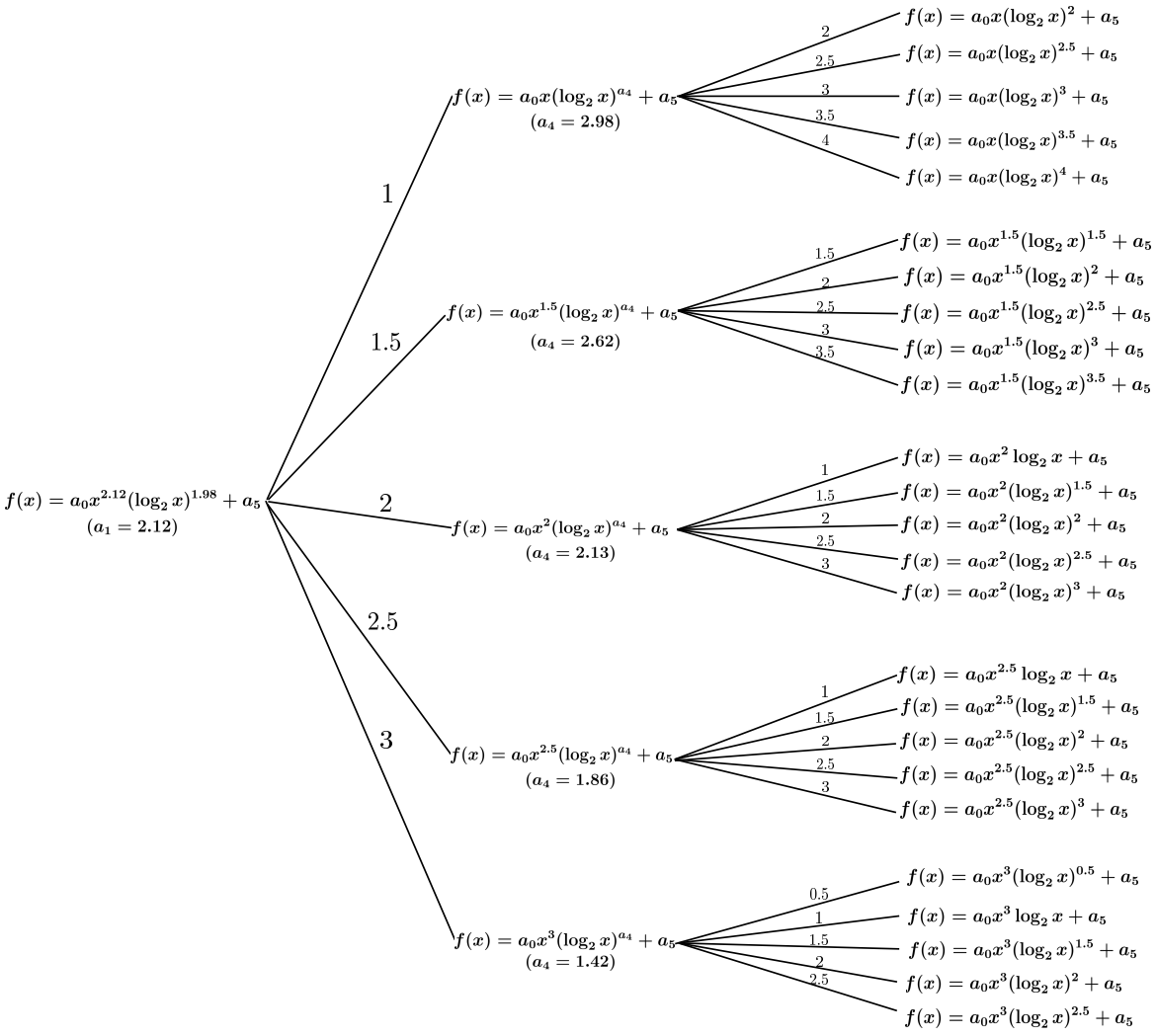
2.5.3 Classificação das funções ajustadas

Após o ajuste de todos os parâmetros das funções em $\mathcal{F}$, cujas funções com parâmetros ajustados estão em $\mathcal{F}_{\text{ajustadas}}$, o EMA as reporta classificadas em cinco grupos (Figura 5(f)), descritos a seguir.

2.5.3.1 Função de erro mínimo

A função de *erro mínimo* é a função $f_{\min} = \text{argmin}\{erro(f) : f \in \mathcal{F}_{\text{ajustadas}}\}$, onde $\mathcal{F}_{\text{ajustadas}}\}$ é o conjunto de funções ajustadas pelo EMA.

Figura 7 – Discretização dos parâmetros da função $f(x) = a_0 x^{a_1} (\log_2 x)^{a_4} + a_5$.



2.5.3.2 Funções viáveis e inviáveis

Estes dois grupos de funções não desempenham um papel crítico como as funções de erro mínimo, equivalentes e de melhor-palpite. Elas ajudam, contudo, ao usuário analisar um pouco melhor os resultados obtidos, e ajudar a decidi-lo se seria conveniente acrescentar novos experimentos. Assim, preferirmos descrevê-las adiante, na Seção 2.6, como parte das características adicionais do EMA.

2.5.3.3 Funções equivalentes

O EMA possui um parâmetro chamado *limiar de equivalência (le)* para que funções sejam consideradas equivalentes àquela de erro mínimo. Este parâmetro é utilizado para definir a função $f_{\max}(x)$ da seguinte forma: para todo $(x,y) \in D$, temos que

$$f_{\max}(x) = \begin{cases} f_{\min}(x)(1 + le), \text{ se } f_{\min}(x) > y, \\ f_{\min}(x)(1 - le), \text{ caso contrário.} \end{cases} \tag{2.4}$$

Uma função f é dita ser *equivalente* a $f_{\min}$ se $erro(f_{\min}) \leq erro(f) \leq erro(f_{\max})$. Ao final da execução é reportado o conjunto F_{equiv} das funções equivalentes.

2.5.3.4 Função melhor-palpite

A função *melhor-palpite* é aquela que o EMA escolhe dentre as equivalentes como a complexidade assintótica do algoritmo. Para elegê-la, o EMA utiliza o critério de selecionar a função mais simples. Uma função é dita ser *mais simples* que outra quando:

(i) possui menos parâmetros livres (isto é, aqueles que a regressão determinará); se iguais, então aquela que

(ii) possui o menor número de termos (cada função tem de 0 a 3 termos); se iguais, então aquela que

(iii) possui menos parâmetros, sejam eles fixos (isto é, aqueles que foram fixados e que a regressão tratará como constantes) ou livres; se iguais, então aquela que

(iv) possui o maior número de parâmetros com valores inteiros.

Utilizando uma execução do EMA para exemplificar a escolha do melhor palpite, a Tabela 4 lista as funções equivalentes para $le = 0.5\%$ e a Tabela 5 lista os valores dos parâmetros de cada função.

Tabela 4 – Funções equivalentes para $le = 0.5\%$, onde $f_{\min} = f_1$.

i	$\boldsymbol{f_i(x)}$	**Parâmetros**	**erro**$(\boldsymbol{f_i})$
1	$a_0x^{e_2}(\log_2 x)^{e_3} + c_1$	a_0,c_1,e_3,e_2	345333203.084047
2	$a_0x^{2.5}(\log_2 x)^{e_2} + c_1$	a_0,c_1,e_2	2264999400.0785785
3	$a_0x^{e_2} + c_1$	a_0,c_1,e_2	8497422048.509853
4	$a_0x^{2.5}(\log_2 x)^3 + c_1$	a_0,c_1	79757171246.83218
5	$a_0x^{2.5}(\log_2 x)^{2.5} + c_1$	a_0,c_1	3489994321925.3726

Tabela 5 – Valores dos parâmetros da Tabela 4.

i	$\boldsymbol{a_0}$	$\boldsymbol{c_1}$	$\boldsymbol{e_3}$	$\boldsymbol{e_2}$
1	$5.254029297961527 \cdot 10^{-6}$	0.999978350237802	1.7677531992775561	2.6164622843010754
2	$7.49265820910712 \cdot 10^{-7}$	1.00214797213195	-	2.93410398531581
3	0.00010058644309565509	0.999999915821857	-	2.7929660376789327
4	$6.268706668633963 \cdot 10^{-7}$	1.0000000016675399	-	-
5	$2.426005443898134 \cdot 10^{-6}$	0.999999869325496	-	-

Pelo critério de simplicidade, são eliminadas primeiramente f_1, f_2 e f_3 pois são as que possuem maior número de parâmetros livres. Como f_4 e f_5 possuem o mesmo número de termos e mesmo número de parâmetros, a função melhor-palpite é f_4 pois possui o maior número de parâmetros com valores inteiros.

Considere uma reanálise mas para $le = 0.05\%$. As funções equivalentes estão listadas na Tabela 6 e os valores dos parâmetros estão listados na Tabela 5, pois são os mesmos da execução para $le = 0.5\%$.

Tabela 6 – Funções Equivalentes para $le = 0.05\%$, onde $f_{\min} = f_1$.

i	$\boldsymbol{f_i(x)}$	**Parâmetros**	**erro**($\boldsymbol{f_i}$)
1	$a_0 x^{e_2}(\log_2 x)^{e_3} + c_1$	a_0,c_1,e_3,e_2	345333203.084047
2	$a_0 x^{2.5}(\log_2 x)^{e_2} + c_1$	a_0,c_1,e_2	2264999400.0785785
3	$a_0 x^{e_2} + c_1$	a_0,c_1,e_2	8497422048.509853

Note que ao diminuir o valor do limiar de equivalência o número de funções equivalentes diminui, pois a tolerância de erro fica cada vez menor e mais próxima da função de erro mínimo. Para obter a função melhor-palpite, primeiramente elimina-se f_1 por ser a que possui maior número de parâmetros livres. Em seguida, elimina-se f_2 por ser a que possui o maior número de termos. Portanto, a função melhor-palpite é f_3.

Pode-se notar que para um mesmo conjunto de pontos representando o consumo de recursos, a função melhor-palpite pode ser diferente dependendo do valor do parâmetro limiar de equivalência. Quanto maior o seu valor, mais funções são inseridas no conjunto de funções equivalentes.

2.6 Características adicionais

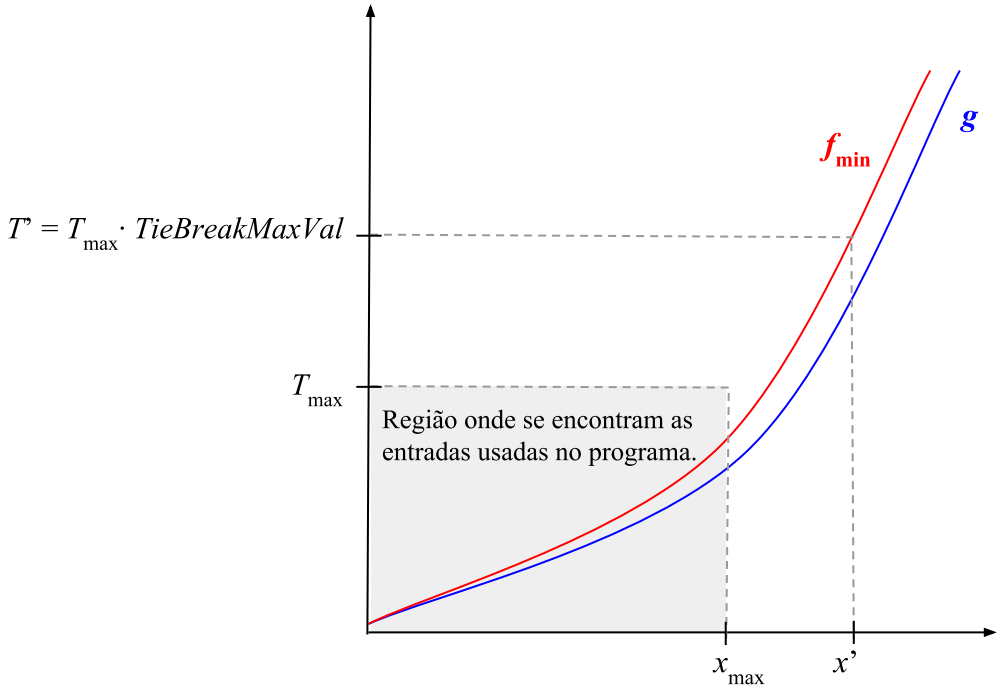
Lembre que um conjunto de funções equivalentes para o EMA representa funções que são muito próximas umas às outras, isto é, dado duas funções do grupo, a diferença entre seus respectivos erros fica limitado a certo parâmetro. A questão que se põe durante a análise é se não seria conveniente aumentar o tamanho das entradas, de modo que a diferença entre parte destas funções aumente de modo que elas deixem de ser equivalentes, melhorando a qualidade do resultado. Neste sentido, o EMA classifica funções como viáveis se dentro de um limite, pode-se crescer a valoração das variáveis de modo a fazer com que o valor da função de erro mínimo e a função em questão se tornem distantes (superior a certo parâmetro), o suficiente para que elas deixem de ser equivalentes. Se uma função for classificada como inviável, então dentro deste limite, mesmo aumentando a variável de entrada, o valor desta função e àquela de custo mínimo não diferiria significativamente.

Formalmente, uma *função viável* é uma função g em que o primeiro ponto x^* tal que $g(x^*)$ e $f_{\min}(x^*)$ diferem de no mínimo certo valor é tal que a execução do programa com entrada x^* requeira tempo/espaço considerado viável. Tal diferença procurada entre as imagens das funções em um mesmo ponto é definido no EMA como 5% (em relação ao valor da imagem de $f_{\min}$). Este ponto, sendo o menor de todos, é chamado de *tieBreaker* de g, informado pelo EMA. Mais formalmente,

$$tieBreaker = \min\{x^* : \frac{|f_{\min}(x^*) - g(x^*)|}{f_{\min}(x^*)} \geq 0.05\}. \tag{2.5}$$

Uma função *inviável* é uma função que não é viável. A Figura 8 mostra graficamente a diferença entre funções viáveis e inviáveis. Seja g uma função a ser classificada entre viável ou inviável. Considere $(x_{\max}, T_{\max})$ o último ponto executado do algoritmo. Considere o tempo calculado como $T' = T_{\max} \cdot TieBreakMaxVal$ (um parâmetro do EMA) e seja x' o ponto onde $f_{\min}(x') = T'$. Considere o menor ponto x^* onde $f_{\min}(x^*)$ difere de $g(x^*)$ em pelo menos 5% em relação ao valor de $f_{\min}(x^*)$. Caso este ponto fique abaixo de x', g é viável de desempatar antes do tempo T'. Caso contrário, g é inviável.

Figura 8 – Classificação de funções entre viáveis e inviáveis.



O EMA limita o número total de processos que executam com o próprio EMA em um computador. Além disso, um novo processo só é executado pelo EMA se existir:

(i) ao menos um núcleo real disponível, para evitar que competição por alocação em CPU afete o tempo de execução.

(ii) memória física disponível, para evitar que a paginação de memória do novo processo em disco aumente seu tempo de execução.

Com respeito ao processamento e geração de entradas, é comum a muitos algoritmos que o tamanho de cada entrada seja de tamanho razoável, como é o caso de algoritmos em grafos, onde as arestas e informações associadas a vértices e arestas precisam ser armazenados. Levando-se em conta que o EMA solicita muitas execuções, a geração temporária de arquivos pode ser um problema operacional se o espaço em disco for insuficiente. Para isto, o EMA controla espaço em disco de forma que cada execução possui uma quota de espaço que, quando ultrapassada faz com que o EMA apague arquivos antigos utilizados como entrada de execuções na área daquele processo.

O EMA pode ser executado em uma máquina que possui sistema operacional Windows ou Linux, tendo como pré-requisitos a instalação local de Python 2.7 e Gnuplot 4.6. A ferramenta está disponível para *download* [1] gratuitamente e é de código-aberto.

3 COMPARAÇÃO DAS FERRAMENTAS DE ANÁLISE AUTOMATIZADA DE ALGORITMOS

Como foi visto no Capítulo 1, encontram-se na literatura referências a algumas ferramentas automatizadas de análise de algoritmos. Ferramentas como ACE [26], ACME [28], ANAC [29] e RAML [45] fornecem a complexidade de algoritmos através da análise do código-fonte, ao passo que o METRIC [25], Trend Profiler [30], Aprof [33], AlgoProf [34], MOCCA [35] e EMA [1] são ferramentas que analisam o algoritmo empiricamente. Também foi visto que a maior parte dessas ferramentas possui muitas limitações, tais como análise de algoritmos implementados em um determinado paradigma de programação e em uma linguagem de programação específica. Além disso, apenas as ferramentas RAML, Trend Profiler, Aprof, AlgoProf e EMA estão disponibilizadas na Internet. Por outro lado, além do EMA, apenas o RAML ainda possui o projeto de pesquisa ativo [37, 38].

Neste capítulo é apresentada uma comparação entre as ferramentas EMA, Trend Profiler, Aprof e RAML. Mais especificamente, algoritmos serão implementados visando a comparação qualitativa dos resultados gerados pelo EMA e por cada uma das outras ferramentas. Os programas para comparação entre EMA e Trend Profiler/Aprof são escritos em C, enquanto entre EMA e RAML são escritos em OCaml. Tal distinção se faz necessária devido a diferença entre linguagens suportadas por cada uma das outras ferramentas sendo comparadas ao EMA. A ferramenta AlgoProf, apesar de estar disponível, não foi comparada com nenhuma outra porque fornece apenas gráficos de desempenho e não uma função de complexidade do algoritmo.

3.1 Trend Profiler

A ferramenta *Trend Profiler* [30] é uma ferramenta de abordagem empírica que descreve o desempenho de um programa em termos de recursos durante sua execução para diversos tamanhos de entrada usando modelos lineares ($y = a + bx$) e de lei de potência ($y = ax^b$). Para determinar a função que descreve a complexidade do algoritmo e seus respectivos gráficos, o Trend Profiler executa regressões lineares aplicadas a escala logarítmica para obter os valores dos coeficientes a e b. É importante ressaltar que a regressão linear minimiza o erro da função na escala logarítmica mas não da função real. O Trend Profiler não determina a complexidade de algoritmos exponenciais ou com termos logarítmicos, mas reporta o gráfico de residuais para que o usuário obtenha conclusões de um possível termo multiplicativo, como por exemplo, um logaritmo.

O Trend Profiler é uma ferramenta de código aberto sem interface gráfica e sua utilização requer a instalação do *gcc*, *gcov*, *perl*, um módulo do perl chamado *Archive::Zip* e o *gnuplot*.

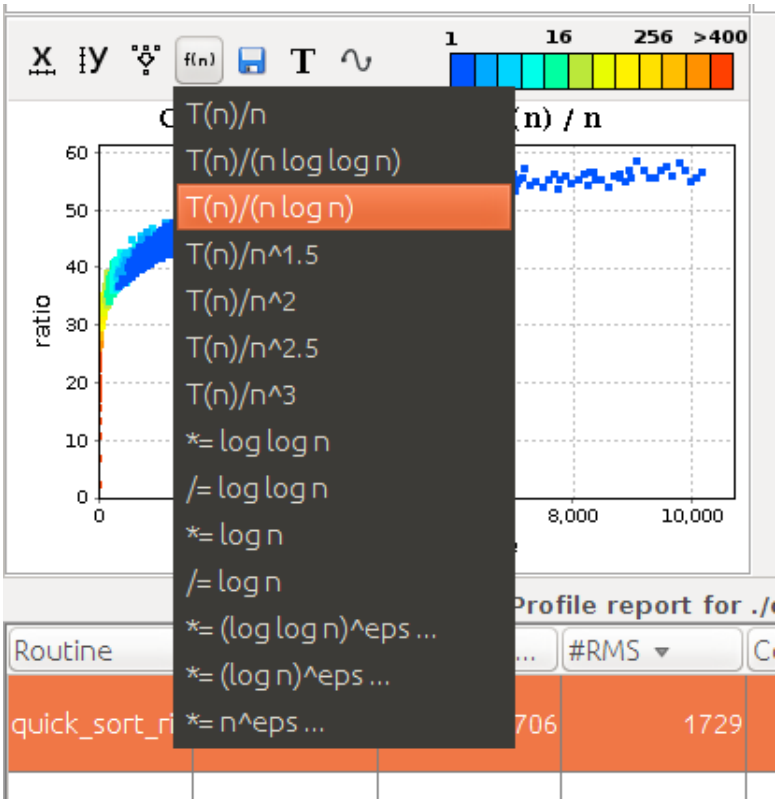
3.2 Aprof

O *Aprof* é uma ferramenta *Valgrind* de abordagem empírica desenvolvida com o objetivo de ajudar os desenvolvedores a descobrirem ineficiências assintóticas ocultas nos

códigos dos programas. Esta ferramenta fornece os custos de desempenho de cada rotina do programa que podem ser usados para produzir gráficos de desempenho e estimar as funções usando técnicas de delimitação de curva. Para fornecer a complexidade de um algoritmo, esta ferramenta utiliza um método para avaliar a taxa de crescimento de uma função da seguinte maneira. Seja n o tamanho de uma entrada, $T(n)$ o custo de execução de n e $H(n)$ uma função de suposição. O Aprof analisa a relação $T(n)/H(n)$. Se $T = \Theta(H)$, então a razão é estabilizada para uma constante positiva, mas caso contrário não se encontra tal convergência. A função de suposição $H(n)$ pertence a uma lista de funções candidatas que deve ser escolhida pelo usuário, podendo também ser uma combinação de monômios, conforme ilustra a Figura 9. Note que funções exponenciais não estão na lista, logo, algoritmos desta natureza não podem ser analisados corretamente pelo Aprof.

O Aprof é uma ferramenta de código aberto e sua utilização requer a instalação da mesma e do *Aprof-plot* que possui uma interface gráfica e serve para gerar os gráficos de desempenho. A instalação do Aprof pode ser feita através de uma imagem *docker* ou diretamente do site. A instalação do Aprof-plot requer o pacote *JDK* previamente instalado.

Figura 9 – Tela da ferramenta Aprof-plot, com a lista de funções candidatas.



3.3 Estudo de caso: BubbleSort

Nesta seção é apresentada uma comparação entre as ferramentas de análise empírica: Trend Profiler, Aprof e EMA através do estudo de caso do BubbleSort implementado em C para entradas de pior caso.

3.3.1 Algoritmo BubbleSort

O *BubbleSort* ou *ordenação por bolha* é um algoritmo de ordenação clássico que consiste em permutar sistematicamente elementos adjacentes que estão fora de ordem [5]. O nome do algoritmo é uma analogia a bolhas em um tanque de água que procuram seu próprio nível. O BubbleSort possui complexidade de pior caso $\Theta(n^2)$ (onde n é o número de elementos da lista) e ocorre quando a lista está em ordem inversa daquela que deseja-se ordenar. O BubbleSort implementado em C encontra-se no Apêndice C.

3.3.2 Resultados

O experimento foi executado da seguinte maneira. Como o Aprof é uma ferramenta Valgrind, o código-executável que o Aprof executa possui muito mais *overhead* que uma compilação direta. Para que os tempos de execução fossem comparáveis, encontramos primeiramente dois tamanhos de entrada, cujos tempo de execução t_1 e t_{30} fossem dentro de intervalos desejados (respectivamente, 200 ms e 18 s). Então, escolhemos mais 28 tamanhos de entrada igualmente distribuídos entre tais dois tamanhos, e todos estes valores foram usados para se gerar entradas e executadas sob o Aprof. Em seguida, utilizou-se o método de calibração do EMA para escolher 30 pontos entre os mesmos limites de tempo.

A execução do BubbleSort através da ferramenta Aprof reportou os gráficos das Figuras 10 e 11. A Figura 10 representa o tamanho de memória lida (eixo x), que neste caso é o tamanho da entrada, e o número de passos para executar tal entrada (eixo y). É possível observar o comportamento quadrático do gráfico. A Figura 11 ilustra o gráfico obtido após a escolha da função $H(n) = n^2$ e é possível observar que esta é a função que descreve a complexidade do algoritmo pois o gráfico da função $T(n)/H(n)$ convergiu para uma constante $c > 0$. Outras funções $H(n)$ foram testadas, mas seus gráficos não convergiram para uma constante positiva.

A execução do BubbleSort através da ferramenta Trend Profiler reportou os gráficos das Figuras 12 e 13. O gráfico da Figura 12 representa a execução do algoritmo, onde o eixo x é o tamanho do arquivo de entrada (em bytes) e o eixo y é o número de passos. Podemos observar que a linha vermelha é a função obtida pela regressão linear e os pontos em azul, verde e roxo representam as curvas de sua vizinhança até o grau 3. Note que a função encontrada ($x^{1.87}$) não corresponde exatamente a complexidade do Bubblesort, mas é um valor próximo. Note a curva pontilhada em azul (x^2) que se aproxima daquela em vermelho e é a que corresponde a real complexidade do Bubblesort. A Figura 13 ilustra o gráfico de residuais do BubbleSort, reportado pelo Trend Profiler. É possível notar que provavelmente não existe algum termo multiplicativo a mais na função (por exemplo, um logaritmo) pois os pontos estão distribuídos em torno de 0 no eixo x.

Figura 10 – Gráfico de execução BubbleSort através do Aprof.

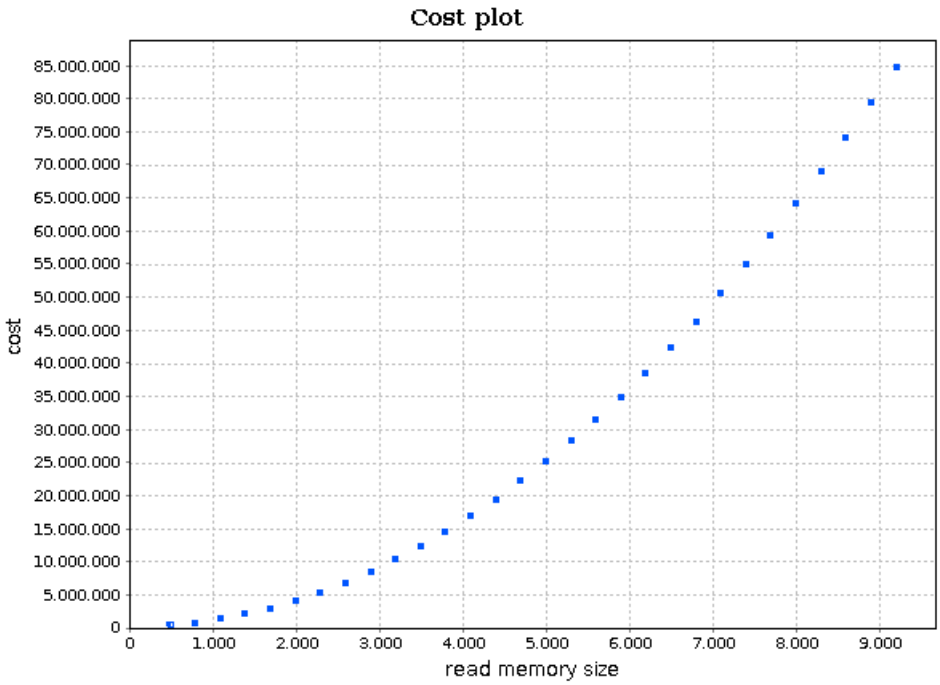
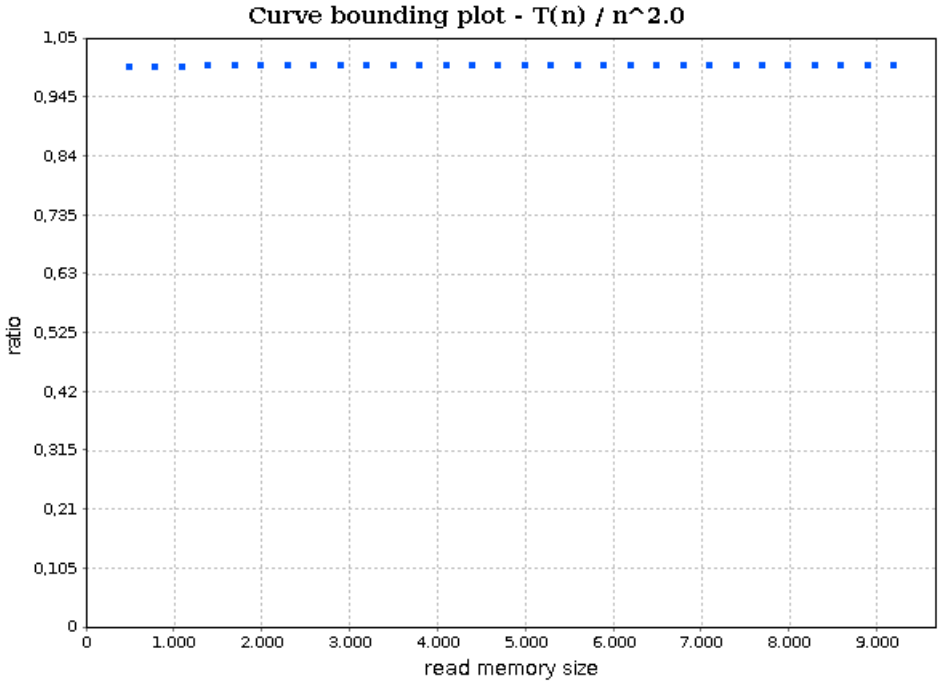


Figura 11 – Gráfico de convergência do BubbleSort para uma constante positiva através do Aprof.



O gráfico reportado pelo EMA ilustrado na Figura 14 e a função encontrada corresponde estritamente ao pior caso do Bubblesort devido as características de sua metodologia que privilegia funções mais simples, conforme os critérios descritos no Capítulo 2. Portanto, o EMA obteve um resultado preciso e de forma automática. O Trend Profiler não fornece uma função precisa que determina a complexidade do algoritmo, ficando por conta do usuário escolher entre a função encontrada pela regressão e uma de sua

vizinhança. O Aprof necessita que o usuário selecione uma função ou combinação de monômios até que observe a convergência do gráfico.

Figura 12 – Gráfico de execução do BubbleSort através do Trend Profiler.

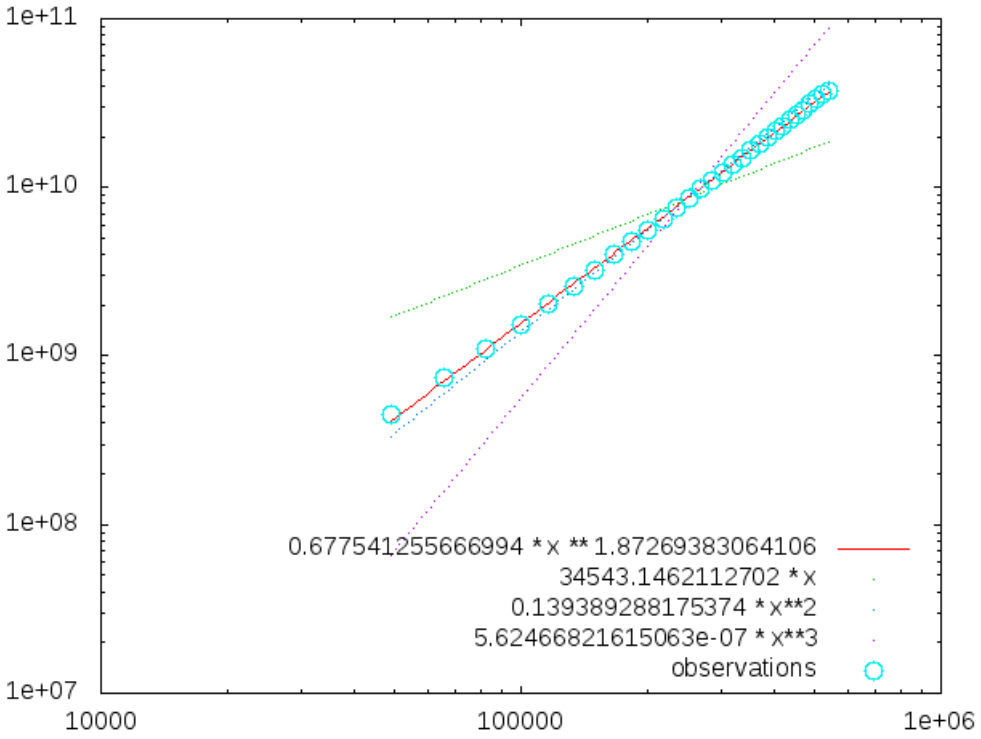


Figura 13 – Gráfico de residuais do BubbleSort através do Trend Profiler.

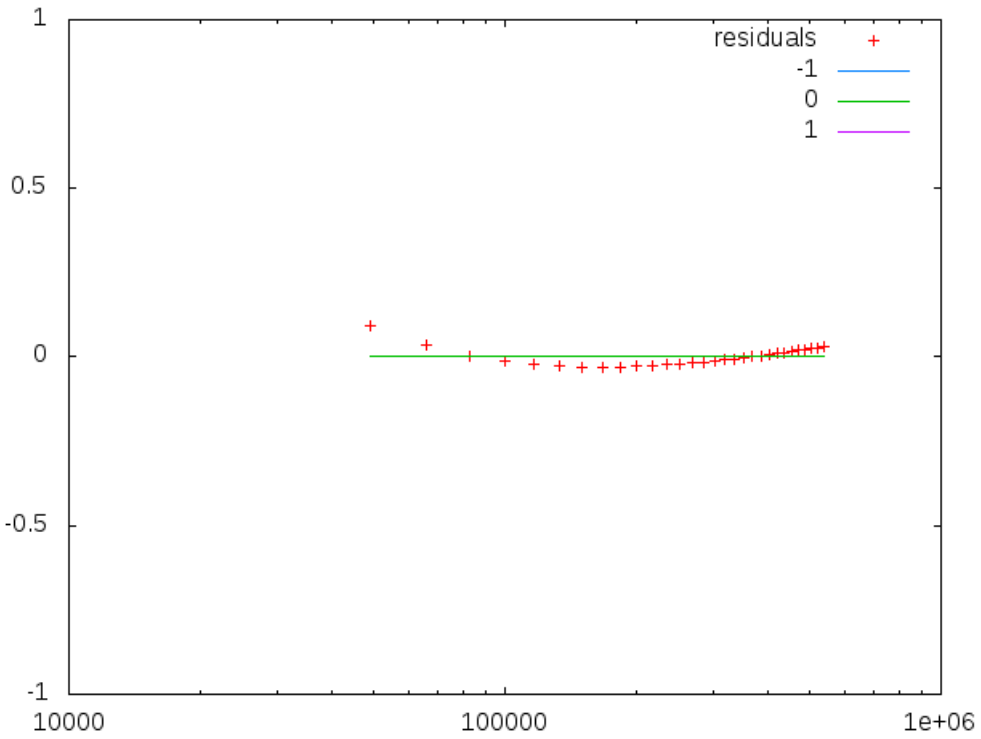
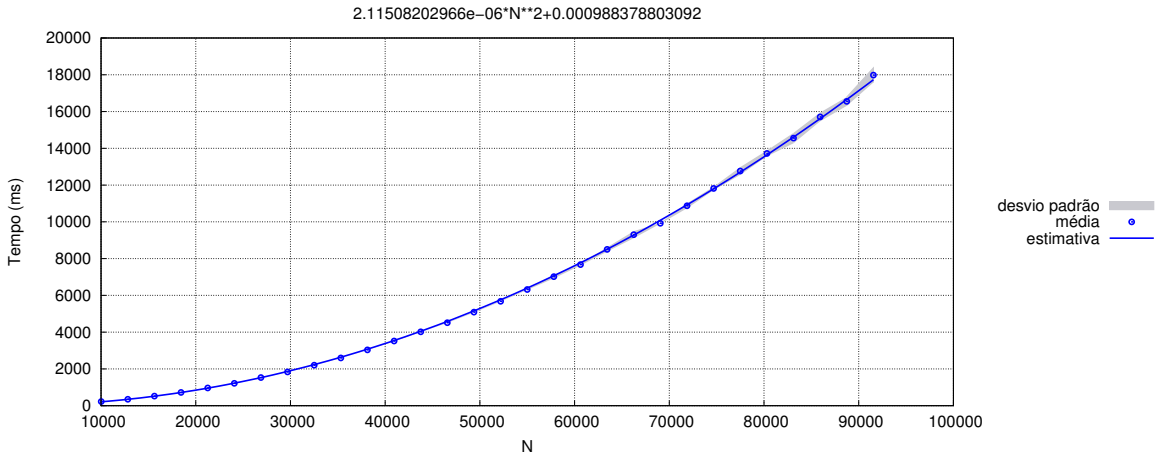


Figura 14 – Gráfico de execução do BubbleSort através do EMA.



3.4 Estudo de caso: MergeSort

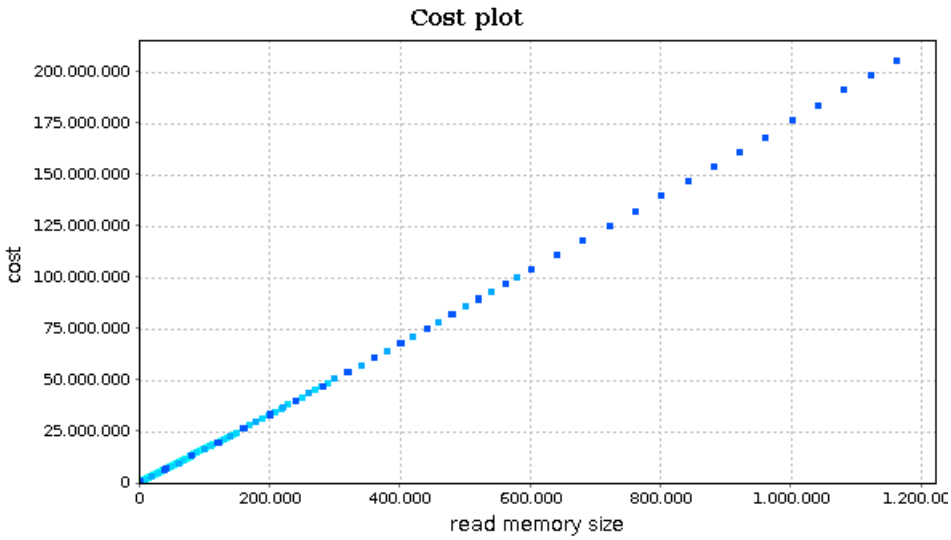
Nesta seção é apresentada uma segunda comparação entre as ferramentas de análise empírica: Trend Profiler, Aprof e EMA através do estudo de caso do MergeSort implementado em C. No Capítulo 4 será abordada a metodologia do MergeSort e seu código em C encontra-se no Apêndice C.

3.4.1 Resultados

Os experimentos realizados com o MergeSort foram executados de maneira análoga àqueles feitos com o BubbleSort na Seção 3.3.

O Aprof reportou os gráficos das Figuras 15, 16 e 17. A Figura 15 ilustra o gráfico do custo da execução do algoritmo e seu formato se parece com uma função do tipo $T(n) = \Theta(n)$. As Figuras 16 e 17 ilustram os gráficos de convergência de algumas funções.

Figura 15 – Gráfico de execução do MergeSort através do Aprof.



Neste experimento fica evidente uma limitação da ferramenta Aprof. Conforme descrito na Seção 3.2, após a ferramenta gerar os gráficos, há a necessidade do usuário supor qual função descreve os pontos obtidos através da escolha de um ou mais monômios combinados de uma lista da ferramenta até observar no gráfico a convergência para uma constante $c > 0$. No entanto, pode haver diversas funções compostas por diferentes monômios que podem convergir para uma constante $c > 0$. Para mostrar que isso ocorre, selecionamos a função correta $H(n) = n \log n$, além de duas funções equivalentes obtidas a partir da análise do programa no EMA e suas respectivas funções arredondadas como função $H(n)$. Note que todos os gráficos da Figura 17 parecem convergir para uma constante assim como o gráfico da função correta ilustrado na Figura 16. Portanto, se o usuário não souber previamente a complexidade do algoritmo que está em análise, pode escolher uma combinação de monômios que apesar de convergir para uma constante, não corresponde a real função de complexidade do algoritmo.

Os resultados obtidos pelo Trend Profiler estão ilustrados nas Figuras 18 e 19. Uma limitação do Trend Profiler é não determinar a complexidade de algoritmos que possuem termos multiplicativos logarítmicos, que é o caso do MergeSort. Contudo, como o Trend Profiler reporta o gráfico de residuais, em casos como esse, é possível prever se existe algum termo a mais na função caso os pontos não estejam aleatoriamente distribuídos em torno de 0 no eixo x, formando uma curva que pode representar um fator logarítmico. Portanto, dessa maneira, é possível o usuário prever a complexidade de um algoritmo neste formato, porém, essa previsão é sujeita a falhas pois não é possível saber qual o valor do expoente desse logaritmo. Apesar do termo linear presente no gráfico da Figura 18, o gráfico de residuais desta execução ilustrado na Figura 19 não reportou um comportamento logarítmico, portanto, o Trend Profiler obteve um resultado não satisfatório para o MergeSort.

Figura 16 – Gráfico de convergência do MergeSort ao selecionar a função $H(n) = n \log n$ através do Aprof.

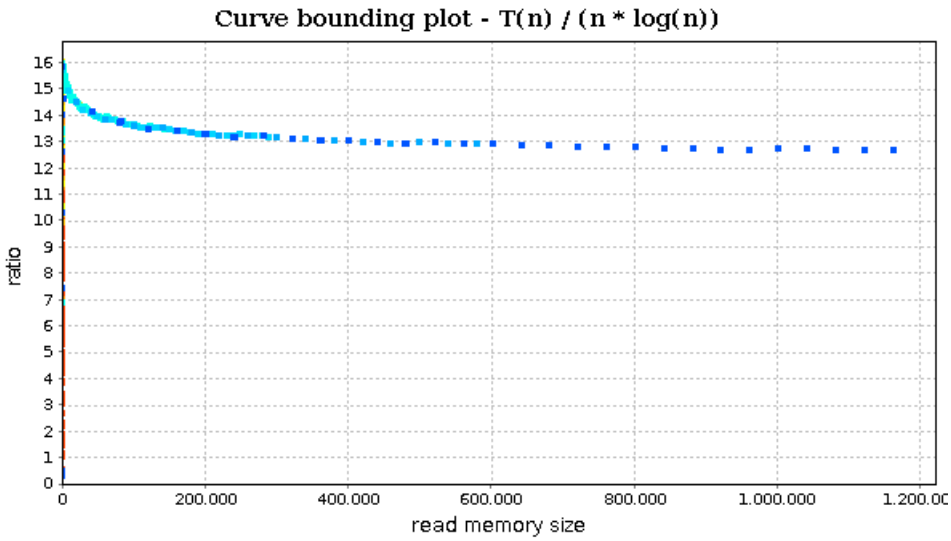
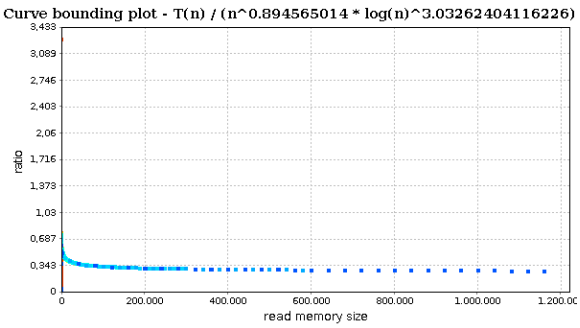
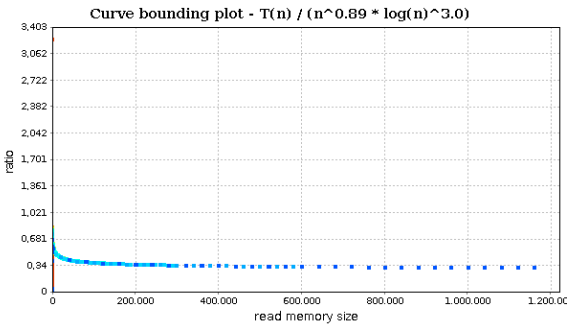


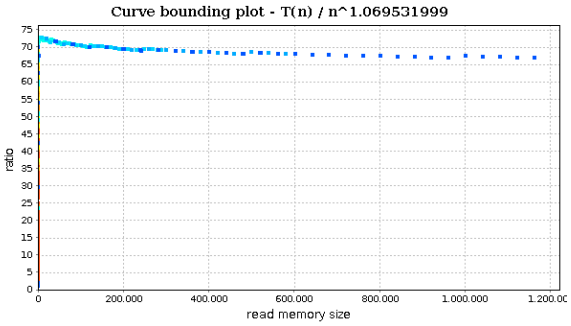
Figura 17 – Gráficos de convergência do MergeSort para uma constante através do Aprof ao selecionar diferentes funções.



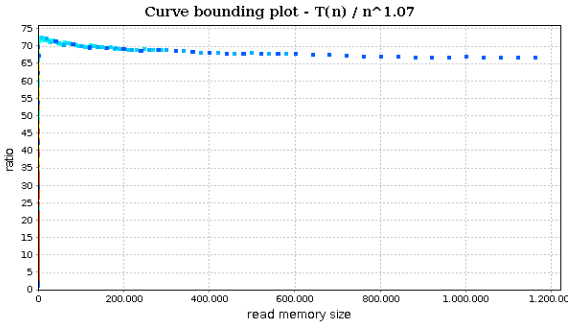
(a) $H(n) = n^{0.894565014}(\log n)^{3.03262404116226}$.



(b) $H(n) = n^{0.89} \log^3 n$.



(c) $H(n) = n^{1.069531999}$.



(d) $H(n) = n^{1.07}$.

Figura 18 – Gráfico de execução do MergeSort através do Trend Profiler.

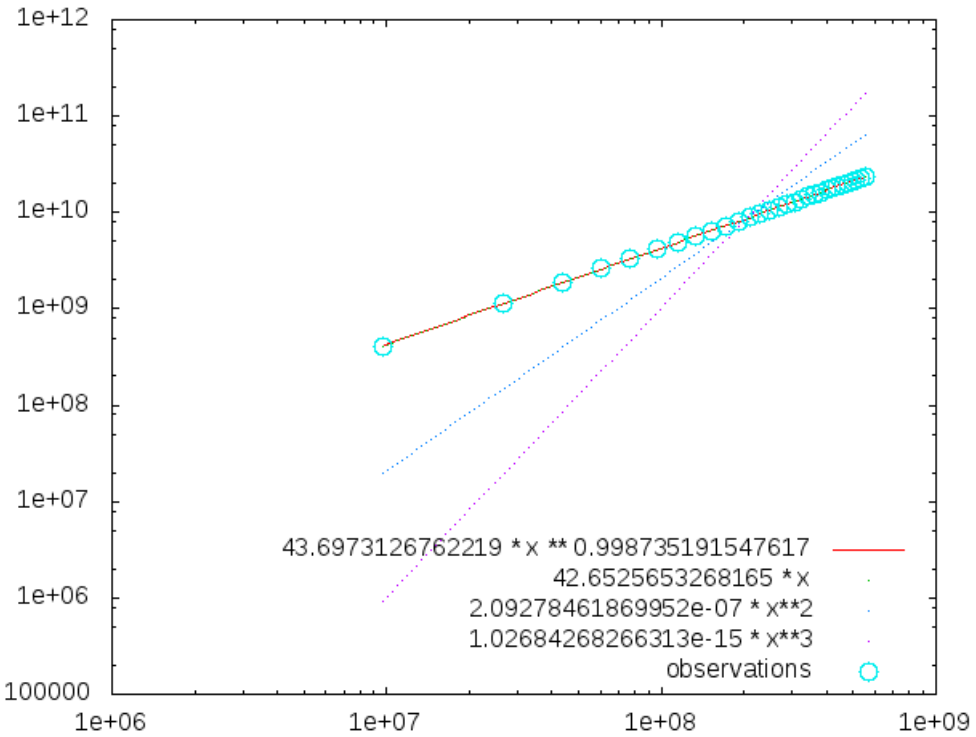
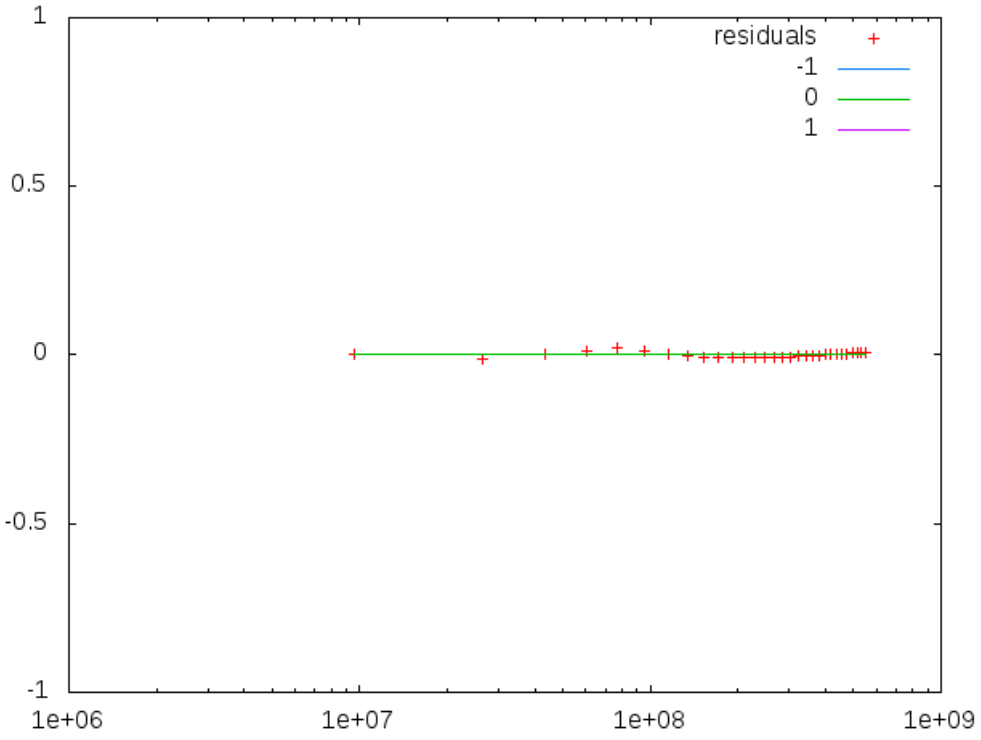
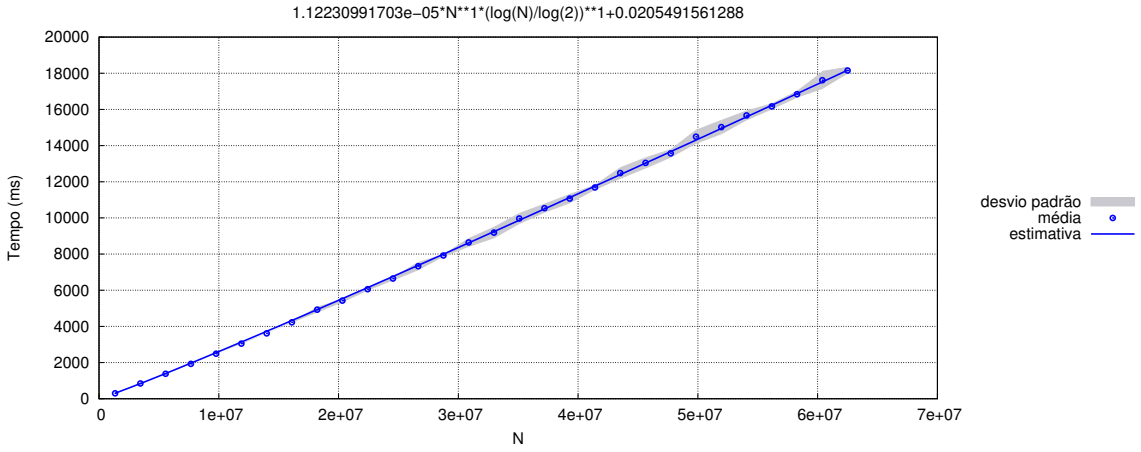


Figura 19 – Gráfico de residuais do MergeSort através do Trend Profiler.



O EMA reportou o gráfico da Figura 20 e observa-se que foi encontrada a função precisa. Ao contrário das ferramentas Aprof e Trend Profiler, o EMA escolhe uma função dentre várias equivalentes sem a necessidade do usuário supor funções que correspondam à complexidade do algoritmo.

Figura 20 – Gráfico de execução do MergeSort através do EMA.



3.5 RAML

O RAML é uma ferramenta com o objetivo de determinar automaticamente funções que estabelecem limites de uso de recursos de algoritmos recursivos escritos na linguagem de programação OCaml. Ele conduz tal processo por análise do código-fonte e não por simulação empírica, e preocupa-se com três métricas do uso de recursos: (i) número de

passos em uma máquina abstrata de alto nível; (ii) células de *heap* alocadas; (iii) número de *ticks* executados que podem ser definidos pelo usuário (um *tick* é um contador de passagem em certa instrução que pode ter um determinado peso). Neste trabalho será usada a métrica associada ao número de passos.

O RAML possui uma plataforma de código aberto com uma interface *web* na qual é possível enviar o algoritmo que deseja-se analisar, além de possuir diversos exemplos prontos que podem ser selecionados, para que o usuário teste a ferramenta em nuvem. Há também a opção de baixar o programa e sua utilização requer a instalação do OCaml 4.01.0 ou 4.02.0, do gerenciador de biblioteca OCaml *Findlib*, das bibliotecas *Jane Street's Core* e *sexplib* e o solucionador de problemas de programação linear CLP (Coin-or Linear Programming).

3.6 Estudo de caso: QuickSort

Nesta seção é apresentada uma comparação entre as ferramentas EMA e RAML utilizando o algoritmo clássico de ordenação *QuickSort*. Como o RAML suporta apenas algoritmos recursivos (o OCaml é uma linguagem de programação funcional purista), para analisar os mesmos programas através do RAML e do EMA foi necessário o uso de um tipo específico de recursão, chamada *recursão de cauda*. Isto se deve à necessidade de aumentar consideravelmente o tamanho da entrada nos testes feitos com o EMA para que haja variação significativa no tempo de execução. Tal aumento por consequência aumenta a pilha de execução do programa por conta do aumento do número de chamadas recursivas empilhadas além do seu limite máximo. Como é conhecido, um cuidado necessário à execução prática de uma recursão é justamente que o número de chamadas recursivas se mantenha pequeno, ou caso contrário pode ocorrer estouro de pilha. Transformando o algoritmo recursivo em recursivo de cauda, o compilador de diversas linguagens de programação, entre elas o OCaml, otimizam o código executável gerado, eliminando-se a recursão por um comando iterativo. Assim, foi possível executar exatamente o mesmo código (implementado em OCaml) nas duas ferramentas e comparar seus resultados conforme será detalhado mais adiante.

3.6.1 Recursão de cauda

A recursão em algoritmos é uma técnica em que uma função ou procedimento que resolve um problema pode chamar a si mesmo para resolver subproblemas menores. Esta técnica segue uma abordagem de divisão e conquista na qual um problema é dividido em problemas menores que são resolvidos recursivamente, porém, se os tamanhos dos subproblemas são pequenos o bastante, estes podem ser resolvidos de maneira direta. Por fim essas soluções são combinadas a fim de criar uma solução para o problema em questão [5]. Mais especificamente, um algoritmo recursivo separa o fluxo da execução em dois casos:

(i) Caso base: instâncias do problema que são resolvidas diretamente, sem a necessidade de resolver subproblemas;

(ii) Caso geral: instâncias cuja solução dependem da solução de subproblemas menores que devem recair eventualmente nos casos bases.

A forma geral de um algoritmo recursivo é dada pelo Algoritmo 4.

Algoritmo 4 Forma geral de um algoritmo recursivo.

```
1: função <NOME-DA-FUNÇÃO>(...)
2:     <bloco-comandos>
3:     se (expressão-detecção-caso-base) então
4:         <bloco-comandos-caso-base>
5:     senão
6:         <bloco-comandos-caso-geral>
7:         <NOME-DA-FUNÇÃO>(...)
8:         <bloco-comandos-caso-geral>
```

Um exemplo clássico de recursão é o algoritmo que calcula o fatorial de um número natural n (Algoritmo 5).

Algoritmo 5 Cálculo de n!

```
1: função FATORIAL(n)
2:     se n = 0 então
3:         retornar 1
4:     senão
5:         retornar n*FATORIAL(n-1)
```

A *recursão de cauda* é aquela na qual não existe processamento a ser feito após o término da chamada recursiva, seja ele um bloco de comandos ou uma expressão a ser calculada com o retorno da chamada recursiva. Desta forma, não há necessidade de manter o estado do contexto da chamada corrente no momento da chamada recursiva. Na recursão comum, para cada chamada recursiva executada é necessário guardar a posição p do código onde foi feita a chamada para que continue executando a partir de p assim que retornado o resultado, ou seja, as chamadas são empilhadas e caso haja um número muito grande de chamadas, pode ocorrer estouro de pilha. Em uma recursão de cauda, não é necessário guardar a posição onde foi feita a chamada, visto que a chamada recursiva é a última operação realizada pela função. Assim, é possível executar recursão de maneira tão eficiente quanto executar um código iterativo.

O Algoritmo 5 pode ser reescrito através da técnica de recursão de cauda. A ideia consiste em mudar o objetivo da função para calcular $an!$, ao invés de $n!$, onde a,n são parâmetros. Trata-se de uma generalização da função pois naturalmente pode-se obter $n!$ passando-se $a = 1$ como parâmetro. Mas esta generalização permite justamente que a recursão de cauda possa ser escrita, como mostra o Algoritmo 6.

Algoritmo 6 Cálculo de an! (recursão de cauda).

```
1: função FATORIAL(n, a)
2:     se n = 0 então
3:         retornar a
4:     senão
5:         retornar FATORIAL(n-1, a*n)
```

3.6.2 Algoritmo QuickSort

O QuickSort é um importante algoritmo de ordenação de uma lista de N elementos que utiliza o paradigma de divisão e conquista. A ideia deste algoritmo resume-se em escolher um elemento da lista denominado *pivô* e então, realizar uma operação chamada *partição*, que consiste em dividir a lista em duas sublistas de maneira que os elementos menores ou iguais ao pivô sejam inseridos em uma sublista e os maiores na outra sublista. Ao fim do processo, temos duas sublistas não ordenadas separadas por um pivô que é maior (resp. menor) que qualquer elemento da primeira lista (resp. segunda lista). Após a partição, o QuickSort ordena as duas sublistas recursivamente, terminando a ordenação [46].

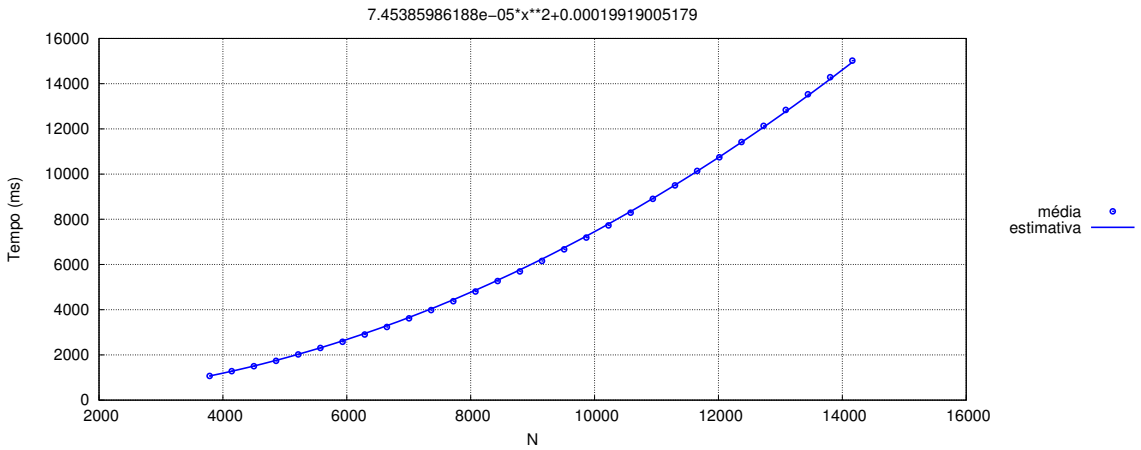
O QuickSort é considerado um dos melhores algoritmos de ordenação na prática devido a sua complexidade média de tempo $\Theta(N \log N)$ a constante multiplicativa implícita pequena. No entanto, seu tempo de pior caso é $\Theta(N^2)$. Para o experimento realizado neste capítulo, o foco é a complexidade de pior caso do QuickSort que ocorre quando o pivô é sempre o maior ou menor elemento da lista, ou seja, quando a lista está ordenada crescentemente ou decrescentemente (assumindo a escolha do pivô sempre o primeiro elemento da lista). Assim, a lista é dividida de maneira o mais desbalanceada possível, isto é, em duas sublistas, sendo uma de tamanho 0 e outra de tamanho $N - 1$ (nossas listas possuirão todos os elementos distintos). Quando isso ocorre em todas as chamadas do método partição, então cada etapa recursiva receberá uma lista de tamanho igual àquele da lista anterior decrementado de uma unidade. A complexidade de pior caso do particionamento é facilmente calculada como sendo $\Theta(N^2)$.

O QuickSort foi implementado na linguagem de programação OCaml utilizando recursão de cauda em todas as funções que compõe o programa, conforme Apêndice C.

3.6.3 Resultados

O experimento apresentado nesta seção consiste em comparar os resultados da análise de complexidade do QuickSort reportada pelo EMA e pelo RAML. Para a análise através do EMA, foram geradas entradas de pior caso e essas entradas tiveram seus tamanhos calibrados também pelo EMA. O resultado da análise do QuickSort através do EMA está ilustrado no gráfico da Figura 21.

Figura 21 – Análise do tempo de execução para entradas de pior caso do QuickSort implementado em OCaml através do EMA.



O mesmo código do QuickSort analisado pelo EMA foi testado pela ferramenta RAML

e o resultado encontrado foi a expressão

$$57.00 - 57.33N + 117.50N^2 + 3.83N^3.$$

É possível observar que a complexidade de pior caso encontrada para o QuickSort através do EMA foi $\Theta(N^2)$. A expressão obtida pelo RAML encontrou complexidade $O(N^3)$, um limite que não é justo. É importante ressaltar que a métrica usada pelo EMA é tempo e pelo RAML é número de passos. Isso explica a diferença entre as constantes multiplicativas encontradas por cada ferramenta.

Além disso, foi realizada no RAML uma análise do QuickSort em sua versão sem cauda, descrito no Apêndice C.

Para esta versão do QuickSort, o RAML obteve

$$3.00 + 15.00N + 14.00N^2,$$

ou seja, $O(N^2)$, que é um limite justo. Portanto, nesse experimento, o RAML obteve um limite superior justo. Com isto, observa-se que o RAML, por ser uma ferramenta de análise de código-fonte, está susceptível ao modo de como o algoritmo está escrito, mesmo que equivalentes do ponto de vista de recursos consumidos.

3.7 Estudo de caso: busca em profundidade

Nesta seção é apresentado um estudo de caso do algoritmo clássico de *busca em profundidade* em um grafo, cuja complexidade é de $\Theta(n+m)$, onde n representa o número de vértices e m aquele de arestas. O objetivo deste estudo é comparar os resultados obtidos pelas ferramentas EMA e RAML.

3.7.1 Algoritmo busca em profundidade

O algoritmo busca em profundidade em um grafo ocupa um lugar de destaque na área de grafos por resolver uma grande quantidade de problemas tais como visitar todas as arestas e/ou vértices, conectividade, detecção de ciclos, determinação de componentes conexas, caminho mínimo entre par de vértices, entre outros. Este algoritmo visa obter um método sistemático de como caminhar pelos vértices e arestas de um grafo, de maneira que dentre todos os vértices visitados e incidentes a alguma aresta ainda não visitada, escolhe-se aquele mais recentemente alcançado na busca [6]. A implementação da busca em profundidade em OCaml está descrita no Apêndice C.

3.7.2 Resultados

O EMA reporta a complexidade assintótica apenas de uma variável por análise, por isso, foram realizados dois experimentos. Em ambos, as arestas foram determinadas aleatoriamente. No primeiro, foi fixado $n = 15\,000$ e variado o valor de m. Portanto, a complexidade do algoritmo esperada é $\Theta(m)$, obtida da expressão geral $\Theta(n+m)$ considerando-se n como constante. No segundo experimento, foi fixado $m = 42\,497$ e variado o valor de n. A complexidade empírica esperada é $\Theta(n)$ por raciocínio análogo. As Figuras 22 e 23 ilustram o resultado reportado pelo EMA nas duas análises. O RAML não foi capaz de analisar este algoritmo.

Figura 22 – Análise da busca em profundidade com $n = 15\,000$.

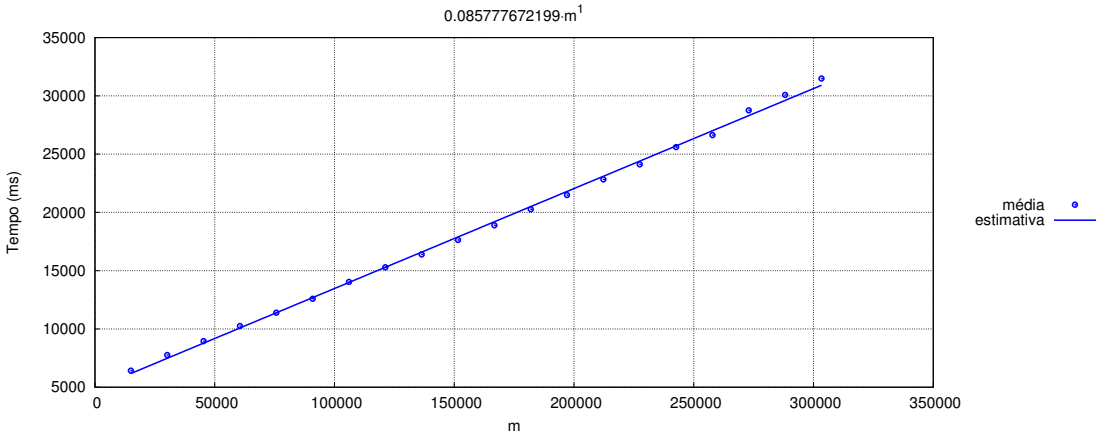
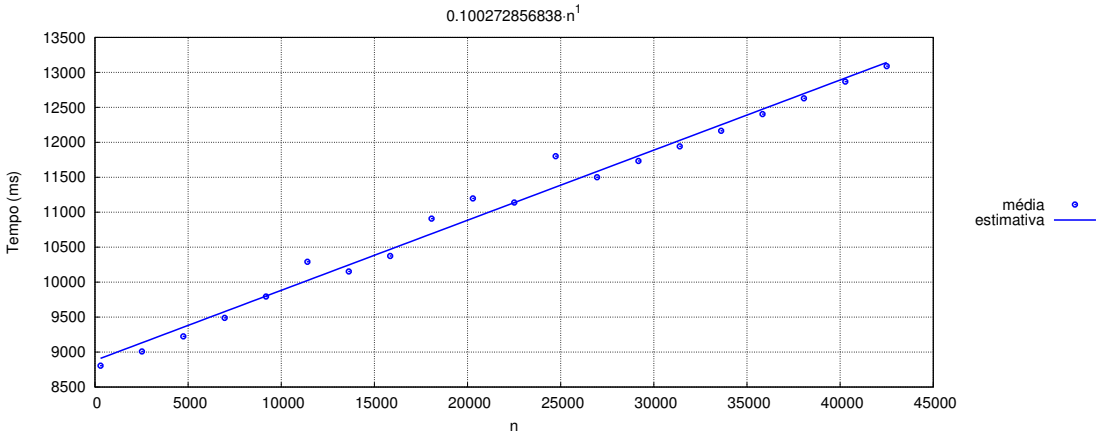


Figura 23 – Análise da busca em profundidade com $m = 42\,497$.



3.8 Considerações finais

Neste capítulo foram realizados dois estudos de caso a fim de comparar três ferramentas de análise empírica: EMA, Trend Profiler e Aprof, utilizando os algoritmos BubbleSort e MergeSort, escritos em C, por ser a única linguagem suportada pelo Trend Profiler e Aprof. No primeiro estudo de caso foi feita uma análise de pior caso do BubbleSort. O Aprof reporta o gráfico de execução de $T(n)$ e é necessário que escolha-se uma função $H(n)$ para que o gráfico de $T(n)/H(n)$ convirja para uma constante $c > 0$. Ao selecionar a função $H(n) = n^2$, pelo gráfico observou-se que a função $T(n)/H(n)$ convergiu, ou seja, o Aprof obteve um resultado justo. O Trend Profiler reportou a função $\Theta(n^{1.87})$ e observou-se no gráfico que dentre os vizinhos de expoentes inteiros, a função $\Theta(n^2)$ também se aproxima dos pontos. Portanto, apesar da função reportada não ser um valor justo por ser obtida por um método numérico, o seu vizinho mais próximo reportado é $\Theta(n^2)$, que é um valor justo. O EMA reportou a função $\Theta(n^2)$ que é a função que corresponde exatamente ao pior caso do BubbleSort. No segundo experimento foi utilizado o MergeSort. No Aprof foram selecionadas cinco funções $H(n)$ e através do gráfico observou-se que todas convergiram para uma constante $c > 0$. Como sabemos que a complexidade do MergeSort é $\Theta(n \log n)$ pode-se dizer que o Aprof obteve um resultado justo. Apesar disso, se o algoritmo em análise tivesse complexidade desconhecida não seria possível saber qual das cinco funções

é a real. A execução do MergeSort no Trend Profiler não obteve um resultado justo pois esta ferramenta limita-se apenas a funções polinomiais. Apesar disso, é reportado também um gráfico de residuais que pode ser usado para indicar se há algum fator multiplicativo logarítmico. Contudo, o gráfico de residuais não apresentou um comportamento que pudesse se suspeitar de um outro termo polilogarítmico. Portanto, o Trend Prof não reportou um valor satisfatório.

Os estudos de caso do BubbleSort e MergeSort mostraram que o EMA reporta resultados mais precisos e de maneira mais automática do que as ferramentas Trend Profiler e Aprof. Ambas possuem metodologia em que idealmente é sabida previamente a complexidade do algoritmo. No caso do Trend Profiler, a execução de um algoritmo de complexidade polinomial de expoente não inteiro desconhecida, pode deixar o usuário na dúvida entre o valor obtido da regressão linear e seu vizinho de expoente inteiro mais próximo. No Aprof, pode ocorrer de muitas funções diferentes convergirem para uma constante, o que não ocorre no EMA que através dos critérios utilizados em sua metodologia que seleciona automaticamente uma função e a reporta para o usuário. Esta escolha automática é derivada de regras que passaram por exaustivos testes, nos mais diversos tipos de algoritmo, de variadas complexidades. Outra limitação do Aprof e Trend Profiler é que ambos só executam algoritmos escritos em C, ao contrário do EMA que analisa algoritmos em qualquer linguagem de programação. O Trend Prof analisa apenas algoritmos polinomiais e reporta o gráfico de residuais que pode indicar um termo multiplicativo, como por exemplo, um logaritmo. Contudo, não é possível saber o expoente (potencial) deste logaritmo. Por outro lado, o Aprof analisa apenas algoritmos da classe polinomial e polilogarítmica, inclusive na forma $\log(\log n)$ não suportada pelo EMA.

Além disso, neste capítulo foram realizados outros dois estudos de caso a fim de comparar as ferramentas EMA e RAML utilizando os algoritmos QuickSort e busca em profundidade, escritos em OCaml, por ser a única linguagem suportada pelo RAML. O fato do EMA precisar crescer o tamanho da entrada, a implementação do QuickSort através de recursão expõe o problema de estouro de pilha e gerou a necessidade de otimizar espaço em memória através de recursão de cauda. A análise de pior caso do QuickSort através do EMA reportou a complexidade $\Theta(N^2)$ que é exatamente a complexidade analítica, diferentemente da análise feita pelo RAML que encontrou uma complexidade de $O(N^3)$. No entanto, para o caso de recursões que não são de cauda, o RAML também encontra um limite justo. No segundo estudo de caso foi feita uma análise de pior caso do algoritmo busca em profundidade, que possui complexidade $\Theta(n + m)$. Para tal experimento no EMA, foi fixada uma variável em uma constante e variou-se a outra e obtivemos $\Theta(m)$ (para n fixo) e $\Theta(n)$ (para m fixo), portanto, complexidades coerentes com aquela obtida pela abordagem analítica. O RAML não foi capaz de analisar este algoritmo. Assim, vemos que a técnica de análise de código-fonte está sujeita a como o algoritmo está escrito, mesmo que sua determinação via método analítico seja rigorosamente a mesma.

A vantagem do RAML em relação ao EMA é não existir a necessidade de instalação de nenhum programa, pois basta utilizar a ferramenta que está disponível na plataforma *web*, enquanto o EMA necessita da instalação do Gnuplot 4.6 e do Python 2.7. Além disso, o RAML fornece a expressão completa da complexidade e o EMA retorna apenas o termo de maior crescimento. Apesar disso, o EMA é capaz de analisar programas em qualquer linguagem de programação, em qualquer paradigma de programação e analisa algoritmos que possuem qualquer tipo de complexidade (polinomial, exponencial, polilogarítmica) enquanto o RAML suporta apenas programas escritos em OCaml e os limites derivados desta ferramenta sempre são polinômios de grau limitado a 6.

4 TESTE DO BIG-ENOUGH

Em análise de algoritmos, o objetivo principal é determinar o consumo assintótico de recursos do algoritmo em função da entrada. O método analítico fornece a função que determina sua complexidade, mas não fornece a partir de qual tamanho de entrada N o consumo observado se comporta de acordo com a função encontrada por esta abordagem. De fato, quando a função que expressa precisamente a complexidade do algoritmo possui monômios de constante multiplicativa muito maior que aquela do monômio de maior grau, para valores modestos da entrada tais monômios dominarão o valor da função, e o consumo aparentaria se comportar como tais monômios. Mas à medida que N crescer, ou quando N for suficientemente grande, tais termos assintoticamente menores poderão ser desprezados, o que implica que a função passa, então, a ser aproximada pelo monômio de maior grau. Portanto, para a tarefa de verificar complexidade via método empírico, é crucial saber a partir de qual tamanho de entrada o algoritmo deve ser submetido. Em termos informais, deve-se saber quão grande é grande o suficiente o tamanho de entrada. Até onde sabemos, não se conduziu até o presente momento uma pesquisa neste sentido.

Com o objetivo de obter o valor inicial de tamanho de entrada para o qual verifica-se empiricamente a complexidade analítica, foi desenvolvida uma metodologia que chamaremos de *teste do Big-Enough*. Para executar os testes, usaremos tanto algoritmos sintéticos cuja complexidade de tempo podem ser manipuladas através de parâmetros, quanto algoritmos reais, para verificar até que ponto as conclusões obtidas com o algoritmo sintético podem ser estendidas para algoritmos reais. Apesar de realizarmos os testes com o recurso tempo, esta abordagem do teste do *Big-Enough* pode ser aplicada a qualquer recurso de interesse. É importante ressaltar que apesar de realizarmos os testes utilizando a ferramenta EMA, esta metodologia pode ser usada em outras ferramentas de análise de complexidade a fim de observar aquelas que detectam a complexidade teórica com a menor entrada possível.

4.1 Metodologia

Dados um programa $\mathcal{P}$ e sua complexidade de tempo assintótica conhecida T, esta metodologia consiste em determinar os valores iniciais de tamanho de entrada N_{be} (em referência a "N *big-enough*"), com respectivo tempo de execução igual a T_{be} para o qual ao realizar uma simulação com execuções de entradas todas limitadas ao tamanho N_{be} (e, portanto, com tempos limitados a T_{be}) são suficientes para, via análise de complexidade empírica, determinar a complexidade analítica.

A determinação de N_{be} é conduzida da seguinte forma: (i) primeiramente, determina-se um valor N_{inf} mínimo para o tamanho da entrada que evita que influências extrínsecas ao algoritmo afetem sua complexidade (por exemplo, tempos de execução muito pequenos, entre 0 ms e 200 ms, podem não conseguir ser medidos corretamente por conta da janela de *time-sharing* do sistema operacional). Para tal determinação, dado um valor T_{min} de tempo, N_{inf} é determinado como o menor valor para o qual $T(N_{\text{inf}}) \geq T_{\text{min}}$; (ii) Dado um valor inicial $N \geq N_{\text{inf}}$, escolhemos um conjunto de pontos no intervalo $[N_{\text{inf}}, N]$ e executa-

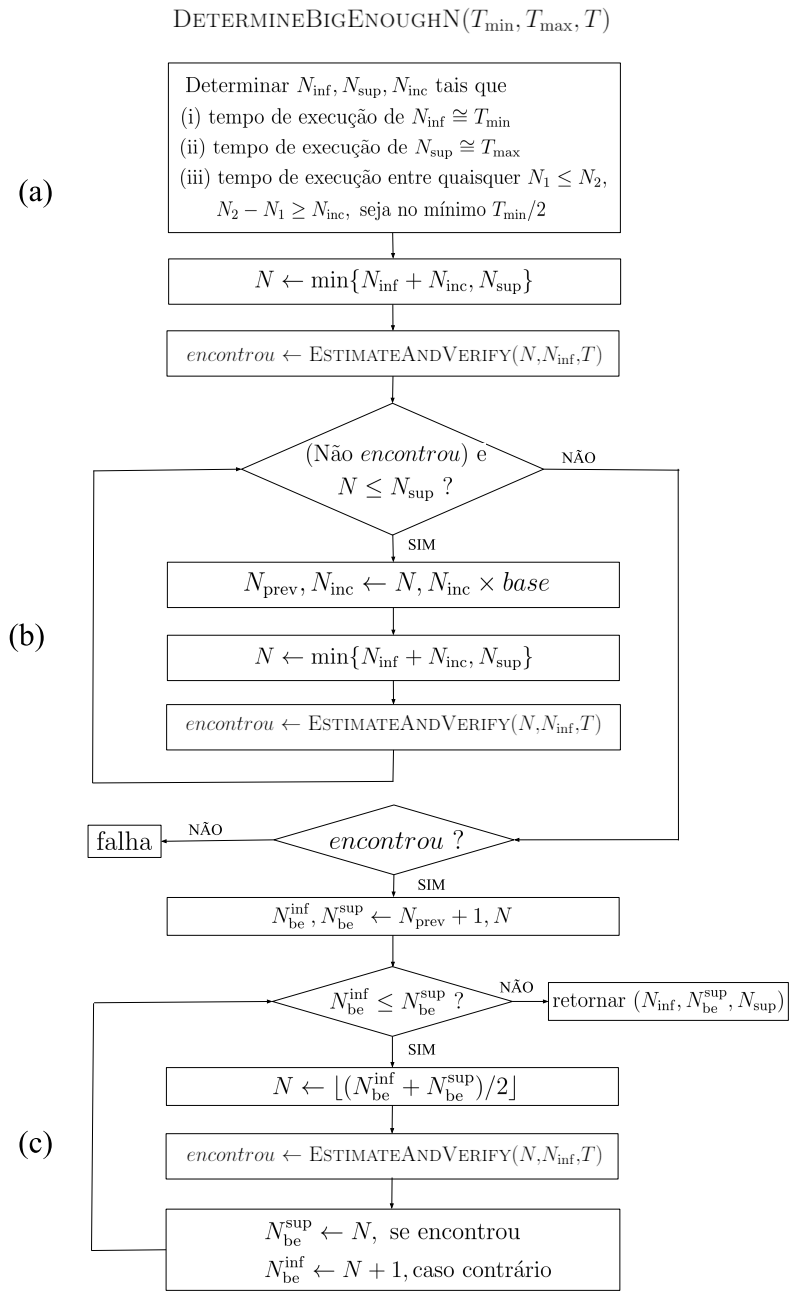
se $\mathcal{P}$ em tais pontos, e ao final detecta-se sua complexidade empírica T'. Enquanto $T' \neq T$, aumenta-se N e reinicia-se o processo. Quando $T' = T$, sabemos que N está em uma região grande o suficiente para poder detectar empiricamente a complexidade analítica. A partir daí, (iii) é feita uma busca binária no intervalo $[N_{\text{be}}^{\text{inf}}, N_{\text{be}}^{\text{sup}}]$, inicialmente dado por $[N_{\text{prev}}, N]$, onde N_{prev} é o último valor de N para o qual ainda não havia sido possível determinar a complexidade, para encontrar de forma mais justa o ponto N_{be} a partir do qual é possível de forma empírica obter a complexidade analítica. Note que o propósito do item (ii) é possibilitar se encontrar rapidamente um limite superior e um inferior para N_{be}, fazendo-se N crescer exponencialmente, para depois N_{be} ser refinado pelo item (iii).

A metodologia do teste do *Big-Enough* descrita acima, em alto nível, foi implementada em Python conforme os Algoritmos 7 e 8. O Algoritmo 7 é descrito alternativamente pela Figura 24.

Algoritmo 7 Teste do *Big-Enough*.

1: **função** DETERMINEBIGENOUGHN($T_{\min}, T_{\max}, T$)
2: Obter $N_{\text{inf}}, N_{\text{sup}}, dN_1, dN_2$ pela calibração do tempo de execução de $\mathcal{P}$ para ficar perto de respectivamente $T_{\min}, T_{\max}, T_{\min} - T_{\min}/4, T_{\min} + T_{\min}/4$
3: $dN \leftarrow dN_2 - dN_1$
4: $N_{\text{inc}} \leftarrow (\text{NUMOFPOINTS} - 1)dN$ ▷ NUMOFPOINTS é uma constante
5: $N \leftarrow \min\{N_{\text{inf}} + N_{\text{inc}}, N_{\text{sup}}\}$
6: $N \leftarrow \text{ARR}(N)$
7: $base \leftarrow 1.3$ ▷ Crescimento exponencial para busca por limite superior de N_{be}
8: $N_{\text{prev}} \leftarrow 0$
9: *encontrou* $\leftarrow$ ESTIMATEANDVERIFY(N,N_{inf},T)
▷ Encontrando limites inferiores / superiores
10: **enquanto** (não *encontrou*) e ($N \leq N_{\text{sup}}$) e ($N_{\text{prev}} < N$) **faça**
11: $N_{\text{prev}} \leftarrow N$
12: $N_{\text{inc}} \leftarrow \lceil base \times N_{\text{inc}} \rceil$
13: $N \leftarrow \min\{N_{\text{inf}} + N_{\text{inc}}, N_{\text{sup}}\}$
14: $N \leftarrow \text{ARR}(N)$
15: *encontrou* $\leftarrow$ ESTIMATEANDVERIFY(N,N_{inf}, T)
16: **se** (não *encontrou*) **então**
17: **retornar** $(N_{\text{inf}}, 0, N_{\text{sup}})$
▷ Encontrando N_{be} (no máximo 5% acima)
18: $N_{\text{be}}^{\text{inf}} \leftarrow N_{\text{prev}} + 1$
19: $N_{\text{be}}^{\text{sup}} \leftarrow N$
20: **se** $N_{\text{prev}} > 0$ **então**
21: **enquanto** $((N_{\text{be}}^{\text{sup}} - N_{\text{be}}^{\text{inf}} + 1)/N_{\text{be}}^{\text{sup}} > 0.05)$ **faça**
22: $N \leftarrow \lfloor (N_{\text{be}}^{\text{inf}} + N_{\text{be}}^{\text{sup}})/2 \rfloor$
23: *encontrou* $\leftarrow$ ESTIMATEANDVERIFY(N,N_{inf}, T)
24: **se** *encontrou* **então**
25: $N_{\text{be}}^{\text{sup}} \leftarrow N$
26: **senão**
27: $N_{\text{be}}^{\text{inf}} \leftarrow N + 1$
28: **retornar** $(N_{\text{inf}}, N_{\text{be}}^{\text{sup}}, N_{\text{sup}})$

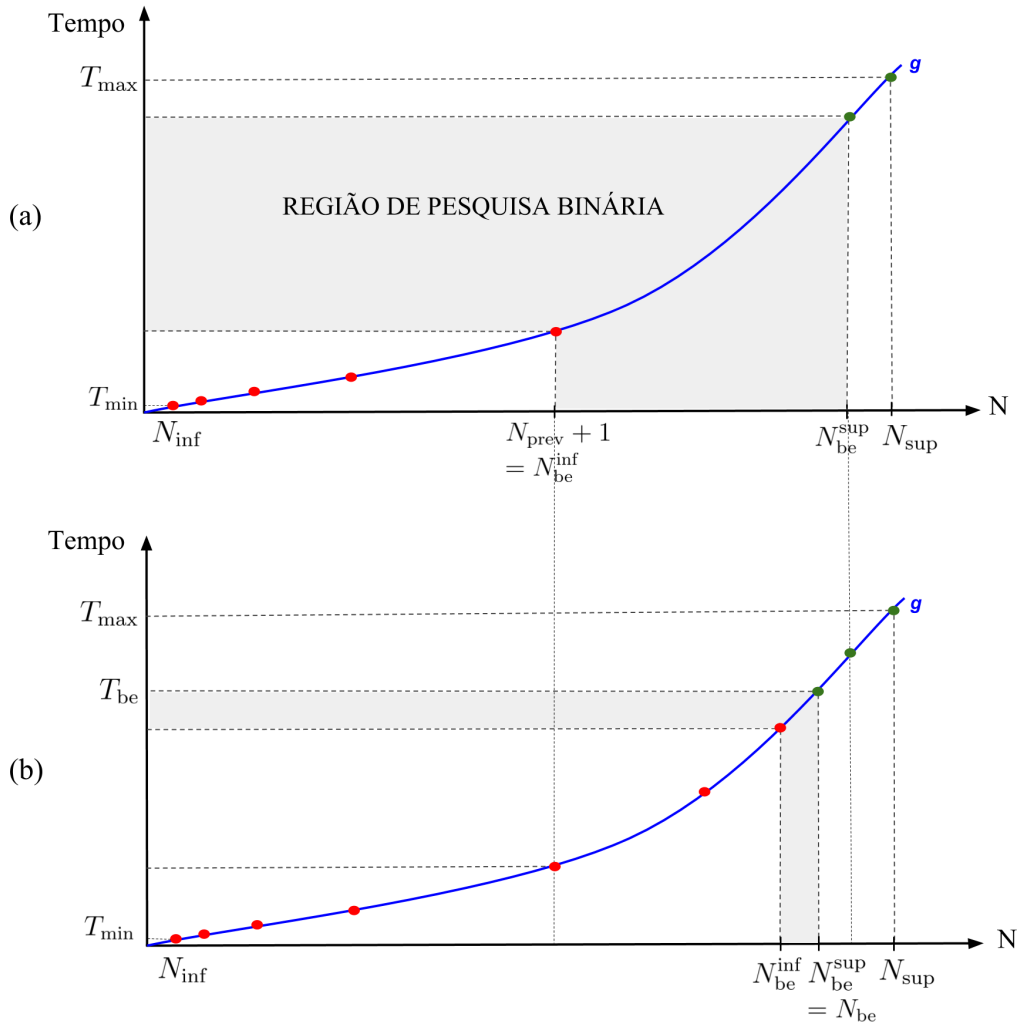
Figura 24 – Metodologia do teste do *Big-Enough.*



O Algoritmo 7 é uma função que retorna os valores:

- $N_{\inf}$: um inteiro tal que $\mathcal{P}$ seja executado em aproximadamente tempo $T_{\min}$. Representa um valor seguro a partir do qual qualquer valor de N executa em um tempo que está livre de atrasos operacionais extrínsecos ao algoritmo (como *time-sharing* do sistema operacional, efeito de cache, entre outros) para o qual devem ser realizados os testes.
- $N_{\be}$: valor de N suficientemente grande, que é o valor procurado pela metodologia.
- $N_{\sup}$: um inteiro tal que $\mathcal{P}$ seja executado em aproximadamente tempo $T_{\max}$ minutos, parâmetro de tempo máximo para a qual devem ser realizados os testes.

Figura 25 – Gráficos de busca da metodologia do teste do *Big-Enough*.



Os valores de N_{inf} e N_{sup} são encontrados através do processo de calibração do próprio EMA (Figura 24(a)), conforme descrito acima. Como exemplo, suponha que queremos obter a complexidade de tempo do InsertionSort, que é $\Theta(N^2)$, e a calibração encontrou $N_{inf} = 13\,769$ e $N_{sup} = 1\,041\,411$ para $T_{min} = 200$ ms e $T_{max} = 30$ min. A primeira busca (item (ii) da Figura 25(a)) tem por objetivo encontrar um limite superior e inferior para N_{be}. Neste ponto, enquanto não for encontrado um valor de N para o qual a complexidade empírica obtida pelo algoritmo, com execuções para tamanhos do vetor de entrada que variam entre N_{inf} e N, é igual a complexidade analítica, guarda-se tal valor de N da iteração anterior (no algoritmo apresentado, em N_{prev}) e incrementa-se exponencialmente o próximo valor de N a ser testado. Para tanto, N receberá o valor de N_{inf} somado a N_{inc}, que é uma variável que crescerá exponencialmente seu valor. A quantidade de tais execuções com tamanhos que variam entre N_{inf} e N é dada por uma constante parametrizada no algoritmo (NUMOFPOINTS). Para os experimentos realizados neste capítulo, utilizamos NUMOFPOINTS = 30.

O valor de dN é calculado de modo que represente que um incremento em N de valor dN garanta um incremento em tempo próximo de T_{min}, que por hipótese é o tempo que elimina os efeitos de problemas na medição (troca de contextos, etc.). Logo, dN poderia ser o próprio valor N_{inf} (já que N_{inf} tem tempo T_{min} e o tempo associado à entrada de tamanho 0 é em geral 0). Porém, se por exemplo, a função crescer mais rápido que

linearmente, podemos dar um incremento em N menor que N_{inf} para obter o mesmo incremento de $T_{\min}$ desejado. Quanto menor o incremento usado, menor (e, portanto, melhor) será o valor final declarado de N_{be}. A heurística para a determinação de dN foi fazê-lo igual à diferença entre os tamanhos de entrada necessários para obter tempos de execução de $3T_{\min}/4$ e de $5T_{\min}/4$. Se a função for linear, tal modo de calcular fará com que dN será o próprio N_{inf}. Caso contrário, será ligeiramente inferior ou superior.

Inicialmente, N_{inc} é dado pelo produto da constante NUMOFPOINTS com dN. Retornando ao exemplo do InsertionSort, suponha que os valores de entrada para os tempos de execução $3T_{\min}/4$ e de $5T_{\min}/4$ foram 12 398 e 15140, respectivamente. Logo, o valor de dN é dado pela diferença entre esses dois valores, portanto, $dN = 2\,742$. O valor inicial de N_{inc}, de acordo com a Linha 4 do Algoritmo 7 é dado por $N_{\text{inc}} = (30-1) \times 2\,742 = 79\,518$. Por outro lado, na repetição da Linha 10 do Algoritmo 7 (Figura 24(b)), esse incremento é obtido pelo produto de N_{inc} por uma *base* (base do crescimento exponencial).

O valor de N a ser testado pelo programa pode ser transformado em um outro valor através da função ARR, que tem por objetivo permitir ao programa sendo testado escolher um outro valor N' o mais próximo de N possível que satisfaça as hipóteses assumidas pelo programa. Por exemplo, se o programa $\mathcal{P}$ sendo testado envolve, digamos, determinar se dado grafo possui emparelhamento perfeito, naturalmente o problema de interesse é que sejam testados apenas grafos com número par de vértices (o número de vértices seria o próprio valor de N), para os quais o algoritmo realmente teria que procurar por emparelhamentos perfeitos ou concluir sua inexistência. Para número ímpar de vértices, o programa poderia parar prematuramente com uma resposta trivial, o que obviamente não está de acordo com o interesse de se testar o pior caso do algoritmo. Como outro exemplo, o algoritmo de Strassen para multiplicação de matrizes necessita de potências de 2 por razões que serão apresentadas mais adiante neste capítulo. Assim, a função ARR transforma valores de N definidos arbitrariamente pelo processo de *Big Enough* em valores próximos que se encaixam melhor nas hipóteses assumidas para a entrada do programa. Se nada for imposto pelo problema em questão, ARR é simplesmente a função identidade.

Por fim, caso a função correta não seja encontrada para nenhum valor de $N \leq N_{\text{sup}}$, o valor de N_{be} retornado é zero. Caso contrário, o algoritmo segue para uma segunda busca (Figura 24(c)), que tem por objetivo encontrar de forma mais justa o valor de N_{be} (Figura 25(b)). Esta etapa consiste de uma busca binária no intervalo $[N_{\text{be}}^{\text{inf}}, N_{\text{be}}^{\text{sup}}]$, inicialmente dado por $[N_{\text{prev}} + 1,\ N]$ (região cinza da Figura 25(a)) obtido pela primeira busca. No exemplo do InsertionSort, o primeiro valor de N testado (Linha 5 do Algoritmo 7) é 93 287. Suponha que para este valor a complexidade empírica não é igual a analítica. Neste caso, o algoritmo executa o primeiro laço de repetição e, assim, temos que $N_{\text{prev}} = 93\,287$, $N_{\text{inc}} = 103\,374$ e $N = 117\,143$. Suponha que para este valor de N a complexidade empírica é igual a analítica. Logo, o próximo passo é executar a busca binária no intervalo $[93\,288, 117\,143]$. O próximo valor de N testado será 105 215, mas para este valor suponha que a complexidade empírica não é igual a analítica. Uma nova busca é feita no intervalo $[105\,216, 117\,143]$ e o próximo valor de N testado será 111 179 e para esse valor a complexidade empírica é igual a analítica. Na Figura 25(b) é possível notar que a pesquisa binária é feita até que N_{be} seja no máximo 5% acima do real valor procurado. Portanto, conforme a Linha 21 do Algoritmo 7 a busca binária segue no intervalo $[105\,2016, 111\,179]$ e o próximo valor de N testado será 108 197 e para este valor a complexidade empírica é igual a analítica. Logo, o algoritmo retornará $N_{\text{be}} = 108\,197$, pois para este valor o critério de parada é satisfeito, ou seja, $(108\,197 - 105\,2016 + 1)/108\,197 = 0.028 < 0.05$.

Algoritmo 8 EstimateAndVerify.

1: **função** EstimateAndVerify(N,N_{inf}, T)
2: Executa $\mathcal{P}(x)$ para $x \in \{N_{\text{inf}} + \lceil (i/\text{NUMOFPOINTS})N \rceil (N - N_{\text{inf}}) : 1 \leq i \leq \text{NUMOFPOINTS}\}$ armazenando o tempo de execução.
3: Seja T' a estimativa empírica da complexidade do tempo de $\mathcal{P}$.
▷ A função empírica e a função analítica são assintoticamente iguais?
4: **retornar** $T' = \Theta(T)$

O Algoritmo 8 descreve a função que retorna verdadeiro se a função encontrada pelo EMA é a função inferida pelo método analítico e falso caso contrário. O conjunto de pontos é gerado de acordo com os valores de N e N_{inf} passados por parâmetro.

Os principais valores de parâmetros do EMA utilizados nas fases de simulação e análise para o teste do *Big-Enough* estão listados na Tabela 7.

Tabela 7 – Valores dos parâmetros do EMA utilizados no teste do *Big-Enough*.

Parâmetro	**Valor**
discardTimeUnder	200
samplingConvergenceFactor	0.005
minNumOfSamples	6
maxNumOfSamples	10
equivalenceThreshold (le)	0.005
tieBreakMaxVal	1.5
case	mean

4.2 Um programa sintético

Nesta seção, aplicaremos o teste do *Big-Enough* em um programa sintético, que pode ser calibrado para que sua complexidade de tempo seja aquela que se deseja. Com efeito, seja $\mathcal{A}$ o programa tal que a sua complexidade de tempo é dada pela função $f(N,a,b,c,d) = a^{N^b} \cdot N^c \cdot (\log_2 N)^d + 200$, onde N é uma variável de entrada e a,b,c,d parâmetros reais não-negativos. Através de valores arbitrários para os parâmetros a,b,c,d é, portanto, possível controlar qual será a complexidade do algoritmo. Mais especificamente, foi implementado o programa $\mathcal{A}$ escrito nas linguagens C e Python, cujo pseudocódigo é mostrado no Algoritmo 9. O programa $\mathcal{A}$ consiste de um procedimento que apenas incrementa uma variável dentro de um laço de repetição que executa $f(N,a,b,c,d)$ vezes. Assim, a complexidade de tempo do programa $\mathcal{A}$ é de fato $\Theta(f(N,a,b,c,d))$.

Algoritmo 9 Programa A.

1: **procedimento** Programa-A(N,a,b,c,d)
2: $z \leftarrow 0$
3: **para** $i \leftarrow 1$ até $f(N,a,b,c,d)$ **faça**
4: $z \leftarrow z + 1$

A Tabela 8 lista os tipos de funções de acordo com os valores de a,b,c,d a serem testados no experimento. Para cada classe, é gerado um conjunto de tuplas a partir do

produto cartesiano $D_a(K) \times D_b(K) \times D_c(K) \times D_d(K)$, onde $1 \leq K \leq 3$ representa a classe de função a ser testada, onde $K = 1$ representa funções polilogarítmicas, $K = 2$ funções polinomiais e $K = 3$ funções exponenciais. Dessa maneira, serão formados todos os valores para os parâmetros a,b,c,d que serão analisados para cada classe.

Tabela 8 – Domínio de f.

$\boldsymbol{K}$	**Classe**	**Domínio de f**			
		$\boldsymbol{D_a(K)}$	$\boldsymbol{D_b(K)}$	$\boldsymbol{D_c(K)}$	$\boldsymbol{D_d(K)}$
1	Polilogarítmica	{1}	{0}	{0}	{5,6,7,8}
2	Polinomial	{1}	{0}	{1,2,3}	{0,1,2,3}
3	Exponencial	{2,3}	{1};	{0,1}	{0,1}

O Algoritmo 10 descreve o método principal do programa, que executa uma função que determina e armazena os valores de N_{inf},N_{be} e N_{sup} para cada função obtida pelo produto cartesiano $D_a(K) \times D_b(K) \times D_c(K) \times D_d(K)$ de cada classe, conforme os valores listados na Tabela 8.

Algoritmo 10 Principal.

1: **procedimento** PRINCIPAL
2: **para** $K \leftarrow 1$ até 3 **faça**
3: **para cada** $(a,b,c,d) \in \mathcal{D}_a(K) \times D_b(K) \times D_c(K) \times D_d(K)$ **faça**
4: N_{inf},N_{be},$N_{\text{sup}} \leftarrow$ DETERMINEBIGENOUGHN$(T_{\min}, T_{\max}, f(N,a,b,c,d))$ onde $T_{\min} = 200$ ms e $T_{\max} = 30$ min
5: GUARDAR(K,(a,b,c,d),N_{inf},N_{be},N_{sup})

Uma primeira análise é a comparação entre as versões dos algoritmos sintéticos em C e Python. A Tabela 9 reporta os valores de N_{be}, T_{be} e it (número de iterações da repetição do Algoritmo 7, Linha 10, necessárias para encontrar N_{be}) obtidos pelo teste do *Big-Enough* para cada função testada pelo programa $\mathcal{A}$. Como a linguagem C é mais eficiente que Python, a constante multiplicativa da função tempo associada a linguagem C é esperada ser menor que aquela associada a um algoritmo equivalente implementado em Python. Além disso, a função tempo do algoritmo em Python geralmente possui outros termos adicionados pela arquitetura da linguagem (como VM, Garbage Collector, etc.). Portanto, espera-se que, em geral, o algoritmo em Python requer uma entrada menor que um algoritmo em C para encontrar a função correta. Logo,

$$N_{\text{be-C}} > N_{\text{be-P}} \tag{4.1}$$

$$T_{\text{be-C}} < T_{\text{be-P}} \tag{4.2}$$

onde os sufixos C e P nas variáveis correspondem a estes parâmetros para a versão em C e Python, respectivamente.

Para encontrar a complexidade correta de algumas funções houve a necessidade de aplicar uma constante multiplicativa c ao termo da complexidade do algoritmo para diminuir o valor de N_{inf} pois algumas funções crescem rapidamente (exponenciais) e outras lentamente (polilogarítmicas), fazendo com que o número de pontos que a metodologia gera não fosse suficiente para detectá-las (encontrando valores de N_{inf} e N_{sup} muito próximos ou iguais). Na implementação em C, aplicamos a constante $c = 10^2$ para f_1 a f_4 e

$c = 10^4$ para f_{18},f_{19},f_{21},f_{22} e f_{23}. Na implementação em Python, aplicamos a constante $c = 10^2$ para f_{19}, $c = 10^6$ para f_{20} e $c = 10^4$ para f_{22} e f_{23}. Observando os resultados de N_{be} para a classe de funções polilogarítmicas (f_1 a f_4) a relação (4.1) não é válida. Isso ocorre porque a constante c foi adicionada apenas à versão implementada em C, porém, os valores de T_{be} estão na mesma ordem de grandeza. Na classe de funções polinomiais, em que nenhuma constante foi adicionada, as relações são estritamente válidas. Já na classe de exponenciais as relações também são válidas para todas as funções, exceto para aquelas em que a constante multiplicativa foi adicionada apenas na versão em C. De maneira geral, podemos observar que apesar dos valores de N_{be} em Python e em C serem um pouco diferentes, pelas razões já explicadas, seus tempos de execução estão na mesma ordem de grandeza.

Tabela 9 – Resultados do teste do *Big-Enough* em funções sintéticas.

i	a	b	c	d	$\Theta(f_i)$	C			Python		
						N_{be}	T_{be}	it	N_{be}	T_{be}	it
1	1	0	0	5	$(\log_2 N)^5$	7 808 702	1 109	1	86 299 478	1 075	2
2	1	0	0	6	$(\log_2 N)^6$	29 112	1 836	1	119 021	1 935	1
3	1	0	0	7	$(\log_2 N)^7$	2 384	38 68	1	4 804	3 411	1
4	1	0	0	8	$(\log_2 N)^8$	532	7 808	1	828	6 430	1
5	1	0	1	0	N	2 512 695 342	4 332	1	48 095 705	3977	1
6	1	0	1	1	$N(\log_2 N)$	95 748 865	4 352	1	2 881 606	5 092	2
7	1	0	1	2	$N(\log_2 N)^2$	5 661 923	4 881	1	219 340	5 659	1
8	1	0	1	3	$N(\log_2 N)^3$	505 964	5 902	1	23 799	6 026	1
9	1	0	2	0	N^2	94 937	15 427	1	13 801	15 718	1
10	1	0	2	1	$N^2(\log_2 N)$	25 495	16 272	1	4 034	16 064	1
11	1	0	2	2	$N^2(\log_2 N)^2$	8 470	20 898	1	1 579	23 070	1
12	1	0	2	3	$N^2(\log_2 N)^3$	2 826	20 582	1	1 042	89 408	3
13	1	0	3	0	N^3	2 950	44 026	1	735	32 656	1
14	1	0	3	1	$N^3(\log_2 N)$	1 357	44 434	1	402	46 712	1
15	1	0	3	2	$N^3(\log_2 N)^2$	644	39 794	1	223	55 540	1
16	1	0	3	3	$N^3(\log_2 N)^3$	823	865 181	5	147	99 832	1
17	2	1	0	0	2^N	36	95 187	1	30	88 214	1
18	2	1	0	1	$2^N(\log_2 N)$	20	78 132	1	28	106 628	1
19	2	1	1	0	$2^N N$	18	81 316	1	19	85 086	1
20	2	1	1	1	$2^N N(\log_2 N)$	-	-	-	6	81 507	1
21	3	1	0	0	3^N	14	82 429	1	19	95 079	1
22	3	1	0	1	$3^N(\log_2 N)$	13	74 161	1	13	3 769 670	1
23	3	1	1	0	$3^N N$	12	109 838	1	9	144 056	1
24	3	1	1	1	$3^N N(\log_2 N)$	-	-	-	-	-	-

A segunda análise é a comparação entre as classes de funções. É possível observar que os valores de N_{be} para as funções da classe polilogarítmica necessitam de valores maiores para que sejam detectadas. Isso se deve ao fato de que a função polilogarítmica cresce mais lentamente que aquelas nas demais classes e para detectar seu comportamento real é necessário um valor maior de N em comparação com outras funções. Na versão em C, esses valores de N_{be} são um pouco menores do que de algumas funções da classe polinomial devido a aplicação da constante c, conforme explicado no parágrafo anterior. Na classe de funções polinomiais, é possível observar que quanto maior o expoente de N e maior o expoente do termo polilogarítmico, menor o valor de N_{be}.

Para as funções em C, apenas a função f_{16} precisou de mais de uma iteração para

ser encontrada. Já na versão em Python, apenas as funções f_1, f_6 e f_{12} precisaram de mais de uma iteração para serem encontradas. Isso ressalta a importância do teste do *Big-Enough* pois algumas funções são mais difíceis de detectar, como aquelas que possuem um termo multiplicativo logarítmico. As funções f_{20} em C e f_{24}, tanto em C quanto em Python, não foram encontradas pelo EMA apesar das tentativas aplicando as constantes multiplicativas 10^2,10^4 e 10^6, além da tentativa de mudar a base do logaritmo para 1.1 no lugar da base 2. Isso se deve ao fato de que para haver variação significativa nos valores de $\log_2 N$, o N deve variar em intervalos maiores, porém, isto torna o termo $2^N N$ muito grande e inviável de ser executado. Logo, em um tempo considerado viável para o teste, o intervalo de N viável de ser usado é pequeno para que o termo polilogarítmico tenha uma variação significativa. O raciocínio é análogo para a função f_{24}.

4.3 Algoritmos reais

Nesta seção apresentamos os resultados do teste do *Big-Enough* aplicado a algoritmos reais conhecidos na literatura que possuem diferentes complexidades de pior caso, implementados na linguagem de programação C e Python para posterior comparação.

Alguns algoritmos apresentados nesta seção precisam de uma atenção especial para que suas funções de complexidade sejam encontradas corretamente. Esses algoritmos são aqueles que são projetados sob o paradigma de divisão-e-conquista. Ao realizar o teste do *Big-Enough* para os algoritmos Strassen e StoogeSort, apresentados mais adiante, projetados sobre tal paradigma, ao utilizar o conjunto de pontos gerados pela metodologia, observamos nos gráficos reportados que os pontos aparecem em formato de escada e torna o trabalho empírico mais difícil. A Figura 26 exemplifica uma execução do algoritmo StoogeSort em que os valores da entrada foram gerados através da metodologia do teste do *Big-Enough* sem o uso da função ARR (Linha 6 do Algoritmo 7). O problema desta configuração em degraus é que ela dificulta o método de regressão a encontrar a função correta. A solução é selecionar apenas os pontos que estão no início de cada degrau através do uso da função ARR. Vale ressaltar que esse problema é agravado em algoritmos que possuem complexidade constante na fase de "conquista", pois o efeito escada é imediato. Por outro lado, algoritmos que possuem complexidade que varia com o tamanho da entrada na fase de "conquista" acrescentam uma complexidade extra a cada patamar à medida que o tamanho da entrada aumenta, suavizando o formato de escada. Deste modo, não foi necessário nenhum tratamento especial para a escolha do conjunto de pontos no caso do MergeSort, por exemplo.

Para exemplificar o aparecimento dos pontos em formato de degrau, considere a busca binária (Algoritmo 11) e uma entrada de tamanho n. Sabemos que seu pior caso ocorre quando o elemento não está na lista e que o algoritmo pára quando $\lceil n/2^i \rceil = 1$, onde i é o número de recursões invocadas. A Tabela 10 lista alguns valores iniciais de n e seus respectivos valores de i. Note que conforme n aumenta, os intervalos de n para cada i também aumentam em patamares (cada intervalo de n na Tabela 10 representa o degrau da escada formada). Portanto, para cada incremento de i é possível notar que o crescimento dos valores do extremo direito dos intervalos crescem exponencialmente, logo, esses são os pontos que serão usados no experimento. O método geral de encontrar tais pontos poderá ser obtido facilmente a partir da discussão na Seção 4.3.6 sobre como encontrar os pontos para o StoogeSort.

Figura 26 – Gráfico de execução do StoogeSort sem o uso da função ARR.

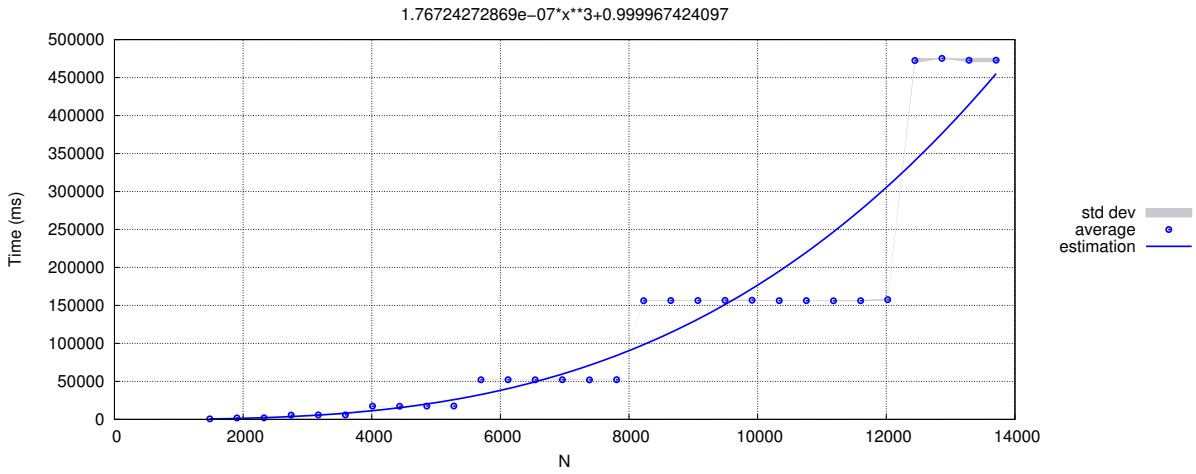


Tabela 10 – Tamanho da entrada (n) e número de divisões (i) da lista em uma busca binária.

n	i
1	0
2	1
[3,4]	2
[5,8]	3
[9,16]	4
[17,32]	5

Algoritmo 11 Busca binária recursiva

```
1: função BUSCABINARIA(L, inicio, fim, x)
2:     meio ← ⌊(inicio + fim)/2⌋
3:     se L[meio] = x então
4:         retornar meio
5:     senão
6:         se L[meio] > x então
7:             retornar BUSCABINARIA(L,inicio,meio − 1,x)
8:         senão
9:             retornar BUSCABINARIA(L,meio + 1,fim,x)
```

4.3.1 Mediana das medianas

Considere o problema chamado de *seleção* que consiste em, dada uma lista L de n números distintos e um inteiro k entre 1 e n, deseja-se obter o k-ésimo menor elemento da lista. A solução trivial para este problema, consiste em ordenar a lista e retornar o elemento da k-ésima posição. Porém, esta solução levaria $\Omega(n \log n)$, utilizando o MergeSort ou HeapSort para ordená-la. O algoritmo conhecido como *mediana das medianas* resolve este problema em tempo linear. Tal tarefa é feita pela divisão da lista L de n elementos em $\lceil n/5 \rceil$ sublistas de cinco elementos cada (o último grupo é o único permitido

possuir menos de 5 elementos). Em seguida, obtém-se a mediana de cada um dos $\lceil n/5 \rceil$ grupos. Recursivamente o algoritmo de seleção obtém a mediana das $\lceil n/5 \rceil$ medianas. Tal elemento, chamado de pivô, é utilizado para separar a lista entre menores (todos em posições abaixo do pivô em L) e maiores que o pivô (todos em posições acima do pivô em L), de modo similar ao feito no QuickSort. Seja m a posição resultante do pivô, após a partição. Se $k \leq m$ o algoritmo de seleção é chamado recursivamente em $L[1..m]$ para encontrar o k-ésimo menor elemento. Caso contrário, o algoritmo de seleção é chamado recursivamente em $L[m+1..n]$ para encontrar o (k-m)-ésimo menor. Para obter a complexidade linear, o algoritmo precisa ter escolhido um "bom pivô" (tal como o QuickSort) e é de fato mostrado que a mediana das medianas cumpre este objetivo de ser tal bom pivô. A metodologia do Mediana das Medianas está descrita no Algoritmo 12.

Pode-se mostrar que a complexidade deste algoritmo é $\Theta(n)$ [5]. Para a implementação em C, o teste reportou $N_{\text{be}} = 26\,653\,266$ e $T_{\text{be}} = 3\,922$ ms. Para a implementação em Python, o teste reportou $N_{\text{be}} = 1\,029\,231$ e $T_{\text{be}} = 4\,954$ ms. As Figuras 27 e 28 ilustram a execução do algoritmo através do EMA referente a última execução do teste do *Big-Enough*.

Figura 27 – Gráfico de execução do Mediana das Medianas em C em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.

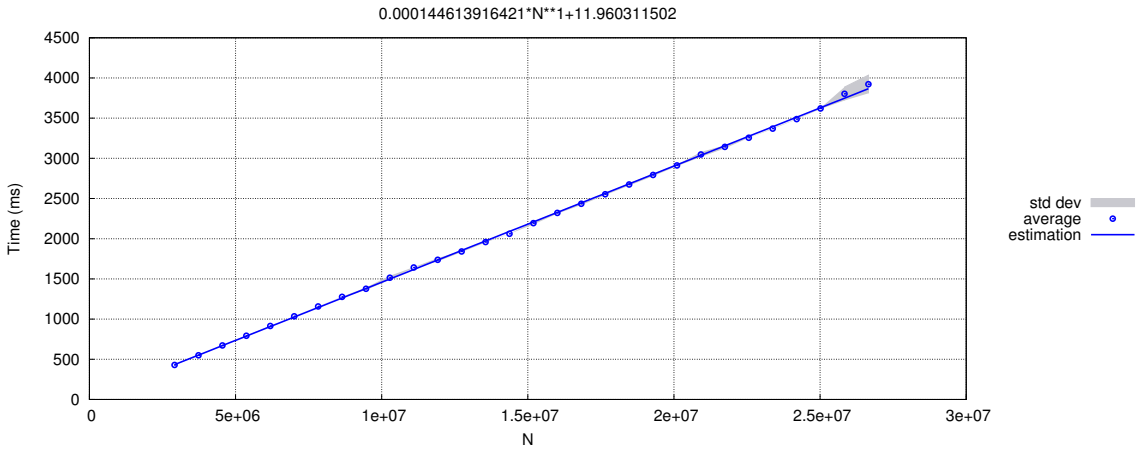
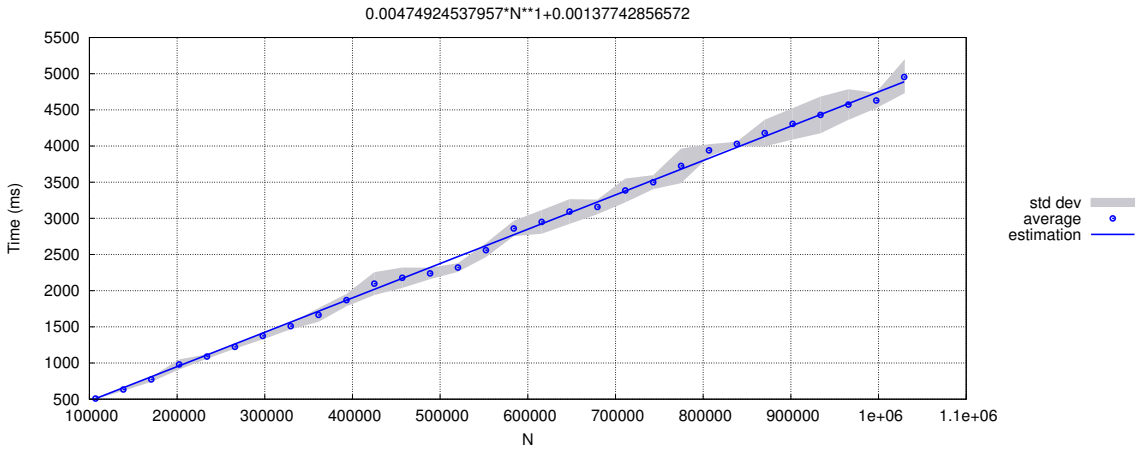


Figura 28 – Gráfico de execução do Mediana das Medianas em Python em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.



Algoritmo 12 Mediana das Medianas

```
1: função SELECIONAR(L,esq,dir,n)
2:     enquanto Verdadeiro faça
3:         se esq = dir então
4:             retornar esq
5:         pivoIndice ← PIVO(L,esq,dir)
6:         pivoIndice ← PARTICAO(L,esq,dir,pivoIndice)
7:         se n = pivoIndice então
8:             retornar n
9:         senão
10:            se n < pivoIndice então
11:                dir ← pivoIndice − 1
12:            senão
13:                esq ← pivoIndice + 1
14: função PARTICAO(L,esq,dir,pivoIndice)
15:    valorPivo ← L[pivoIndice]
16:    L[pivoIndice],L[dir] ← L[dir],L[pivoIndice]
17:    indice ← esq
18:    para i ← esq até dir − 1 faça
19:        se L[i] < valorPivo então
20:            L[indice],L[i] ← L[i],L[indice]
21:            indice ← indice + 1
22:    L[dir],L[indice] ← L[indice],L[dir]
23:    retornar indice
24: função PIVO(L,esq,dir)
25:    se dir − esq < 5 então
26:        retornar PARTICAO5(L,esq,dir)
27:    para i ← esq até dir passo 5 faça
28:        subdir ← i + 4
29:        se subdir > dir então
30:            subdir ← dir
31:        mediana5 ← PARTICAO5(L,i,subdir)
32:        L[mediana5],L[esq + ⌊(i − esq)/5⌋] ← L[esq + ⌊(i − esq)/5⌋],L[mediana5]
33:    retornar SELECIONAR(L,esq,esq + ⌊(dir − esq)/5⌋, esq + (dir − esq)/10)
34: função PARTICAO5(L,esq,dir)
35:    ORDENAR(L)                                   ▷ Qualquer algoritmo
36:    tam ← dir − esq + 1
37:    se tam = 1 ou tam = 2 então
38:        retornar esq
39:    senão
40:        se tam = 3 ou tam = 4 então
41:            retornar esq + 1
42:        senão
43:            retornar esq + 2
```

4.3.2 MergeSort

O *MergeSort* é um algoritmo de ordenação que utiliza a abordagem de divisão e conquista. A ideia deste algoritmo consiste em dividir a lista em duas partes B' e B'' de mesmo tamanho e ordenar cada uma delas. Em seguida, mescla-se B' e B'' da seguinte maneira. Como o menor elemento dentre todos aqueles de B' e de B'' ou é o primeiro elemento de B' ou é o primeiro elemento de B'', verifica-se tal condição e insere tal menor elemento na primeira posição livre de uma lista C. O processo é repetido até que todos os elementos de B' e de B'' estejam em C ordenados. Em seguida, retorna-se os elementos de C para a memória correspondente a B' e a B''. O MergeSort está descrito no Algoritmo 13.

Este algoritmo possui complexidade de pior caso $\Theta(n \log n)$. Para a implementação em C, o teste reportou $N_{\text{be}} = 16\,441\,378$ e $T_{\text{be}} = 4\,559$ ms. Para a implementação em Python, o teste reportou $N_{\text{be}} = 1\,198\,744$ e $T_{\text{be}} = 4\,988$ ms. A Figuras 29 e 30 ilustram a execução do MergeSort em C e Python através do EMA para o teste do *Big-Enough*.

Algoritmo 13 MergeSort

```
1:  procedimento MERGESORT(B,inicio,fim)
2:      se inicio < fim então
3:          m ← (inicio + fim)/2
4:          MERGESORT(B,inicio,m)
5:          MERGESORT(B,m + 1,fim)
6:          MERGE(B,inicio,m,fim)
7:  procedimento MERGE(B, inicio, m, fim)
8:      var C[1..fim − inicio + 1]
9:      i,j ← inicio, limite + 1
10:     para k ← 1 até fim − inicio + 1 faça
11:         se j > fim ou (i ≤ m e B[i] ≤ B[j] então
12:             C[k],i ← B[i],i + 1
13:         senão
14:             C[k], j ← B[i], j + 1
15:     B[inicio..fim] ← C[1..fim − inicio + 1]
```

Figura 29 – Gráfico de execução do MergeSort em C em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.

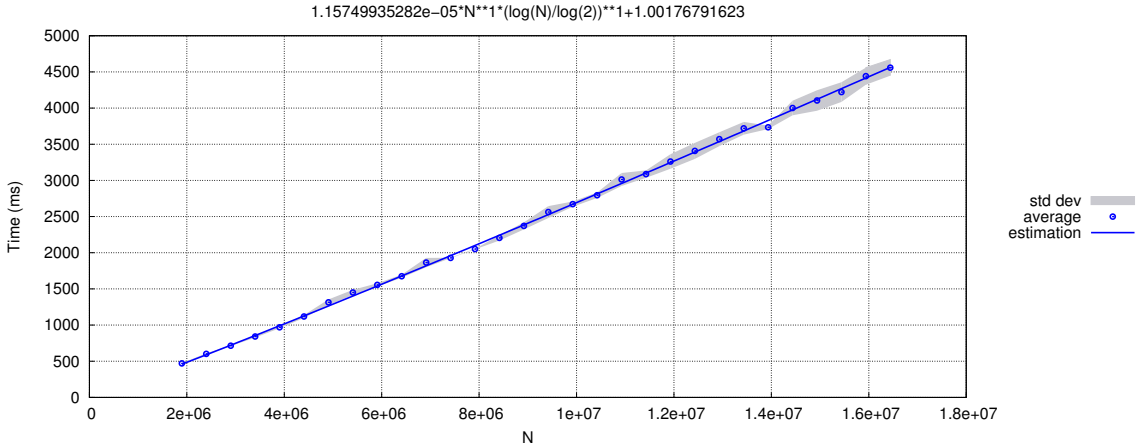
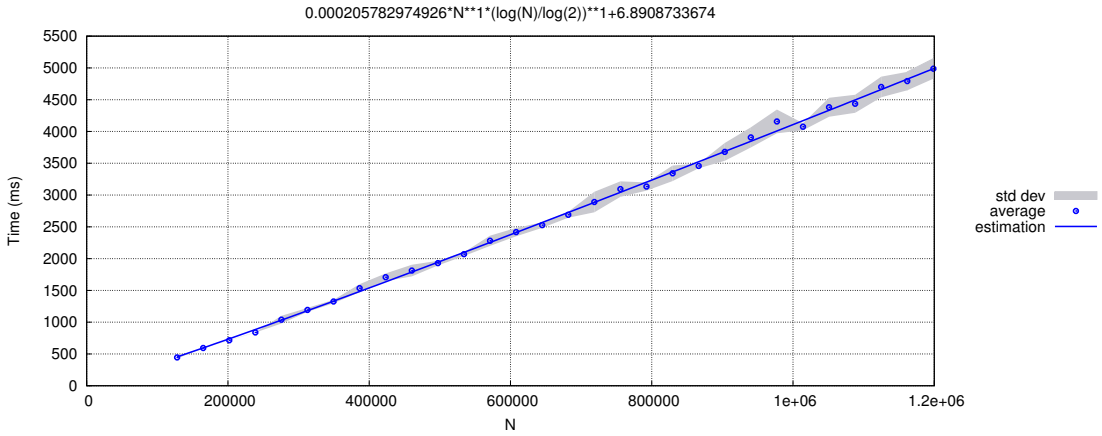


Figura 30 – Gráfico de execução do MergeSort em Python em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.



4.3.3 MergeSort long

Sabemos que a complexidade do algoritmo MergeSort é $\Theta(n \log n)$, porém, isso só é válido se cada elemento da lista cabe em uma palavra de máquina (que corresponde a uma variável) pois o custo de uma atribuição a uma variável e aquele de comparação de valores entre duas variáveis é unitário (modelo de custo unitário – ver Introdução). Mas suponha que deseja-se ordenar um vetor de inteiros, porém, esses inteiros são maiores do que uma variável do tipo inteiro pode armazenar (ou seja, maiores que uma palavra de máquina). Para resolver este problema, desenvolvemos uma versão do MergeSort para ordenar inteiros longos. Para a implementação em C, em vez de utilizar um vetor de inteiros, utilizamos um vetor de estruturas que contém um campo *dig* do tipo vetor de inteiros, na qual cada posição de *dig* armazena um dígito binário de tal inteiro. Para a implementação em Python, utilizamos uma lista L onde cada elemento é uma lista L_{dig} e cada posição de L_{dig} contém um dígito binário do inteiro representado em L. Portanto, o custo de atribuição/comparação de uma variável passa a ser $O(\log n)$ no valor sendo atribuído/comparado, pois todos os dígitos da lista que representa o elemento devem ser acessados, no pior caso. Dada essa alteração, a complexidade do MergeSort para inteiros longos é $\Theta(n \log^2 n)$.

Figura 31 – Gráfico de execução do MergeSort long em C em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.

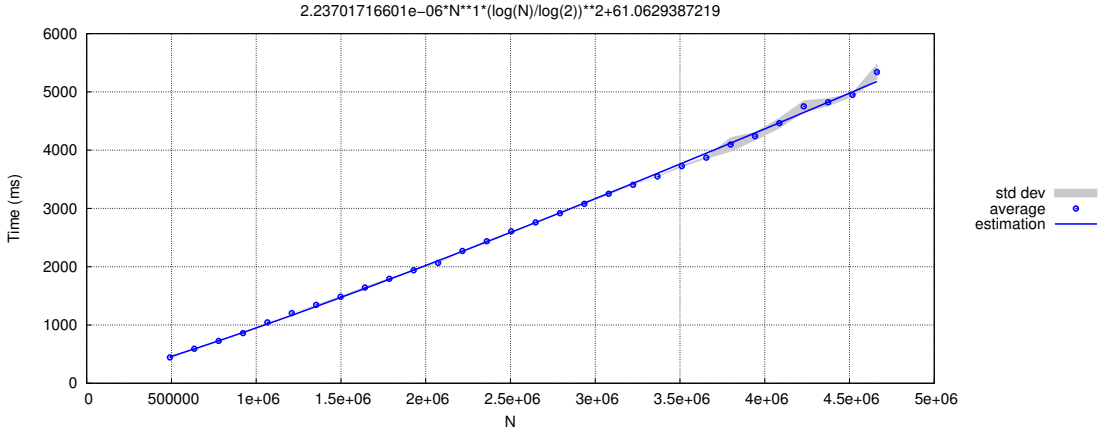
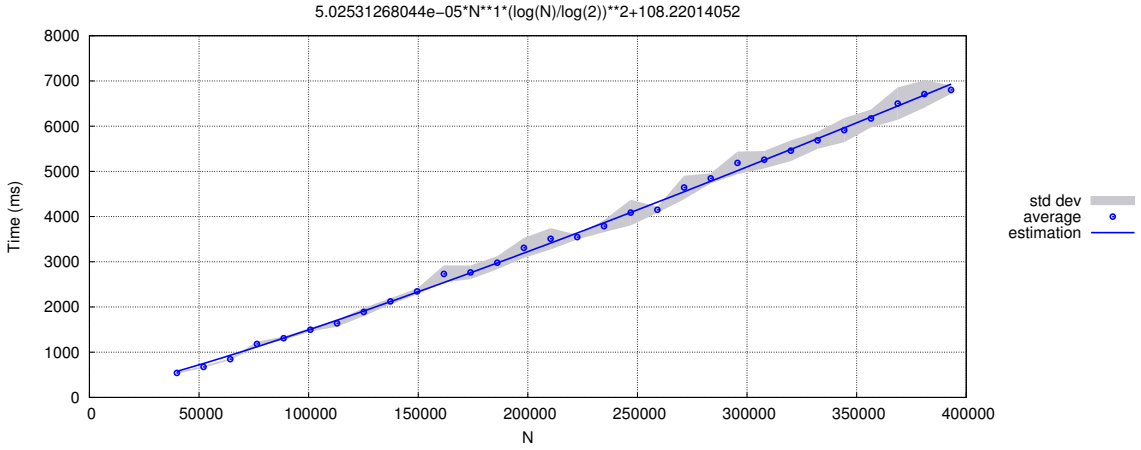


Figura 32 – Gráfico de execução do MergeSort long em Python em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.



Para a implementação em C, o teste reportou $N_{\text{be}} = 4\,661\,611$ e $T_{\text{be}} = 5\,341$ ms. Para a implementação em Python, o teste reportou $N_{\text{be}} = 393\,088$ e $T_{\text{be}} = 6\,802$ ms. As Figuras 31 e 32 ilustram o gráfico da execução do algoritmo nas linguagens C e Python através do EMA para o teste do *Big-Enough*.

4.3.4 InsertionSort

O *InsertionSort* é um algoritmo de ordenação que utiliza uma técnica muito semelhante àquela usada para ordenar as cartas de um baralho. Supondo que as cartas estão ordenadas e uma nova carta é recebida, esta deve ser inserida na sua posição correta, de maneira que todas as cartas permaneçam ordenadas. A cada nova carta recebida, o processo se repete. Esta é uma ideia análoga ao que acontece no InsertionSort: considera-se que o primeiro elemento está ordenado, isto é, na posição correta, o que é naturalmente verdade. A partir do segundo elemento, os demais elementos são inseridos na posição apropriada entre aqueles já ordenados. O elemento é inserido na posição adequada movendo-se todos os elementos maiores para a posição seguinte. O InsertionSort está descrito no Algoritmo 14.

O InsertionSort possui complexidade de pior caso $\Theta(n^2)$ e ocorre quando a lista está com os elementos em ordem inversa daquela ordenada. Para a implementação em C, o teste reportou $N_{\text{be}} = 93\,287$ e $T_{\text{be}} = 14\,479$ ms. Para a implementação em Python, o teste reportou $N_{\text{be}} = 17\,742$ e $T_{\text{be}} = 15\,919$ ms. As Figuras 33 e 34 ilustram a execução do InsertionSort em C e Python através do EMA para o teste do *Big-Enough*.

Algoritmo 14 InsertionSort

```
1: procedimento INSERTIONSORT(B, N)
2:     para i ← 2 até N faça
3:         j ← i − 1
4:         enquanto j > 0 e B[j] > B[j + 1] faça
5:             B[j],B[j + 1], j = B[j + 1], B[j], j − 1
```

Figura 33 – Gráfico de execução do InsertionSort em C em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.

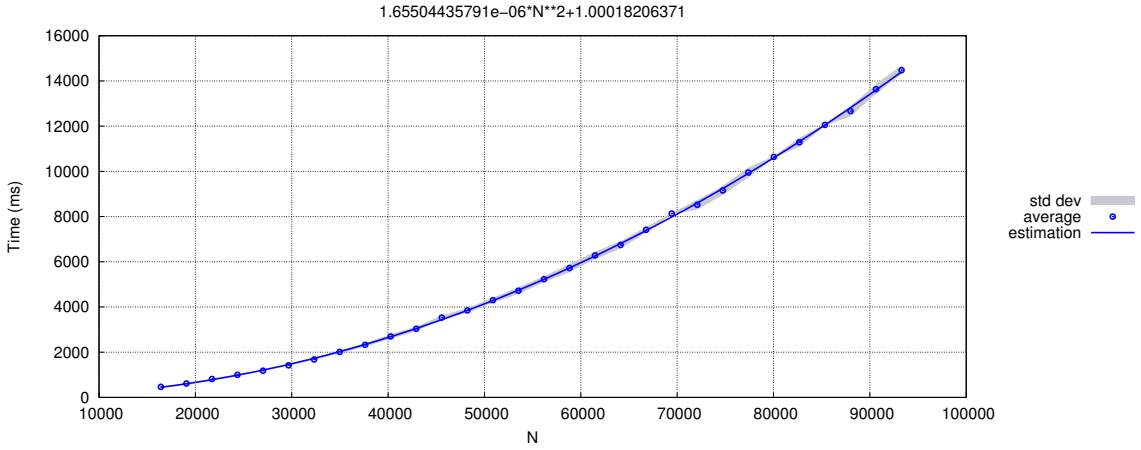
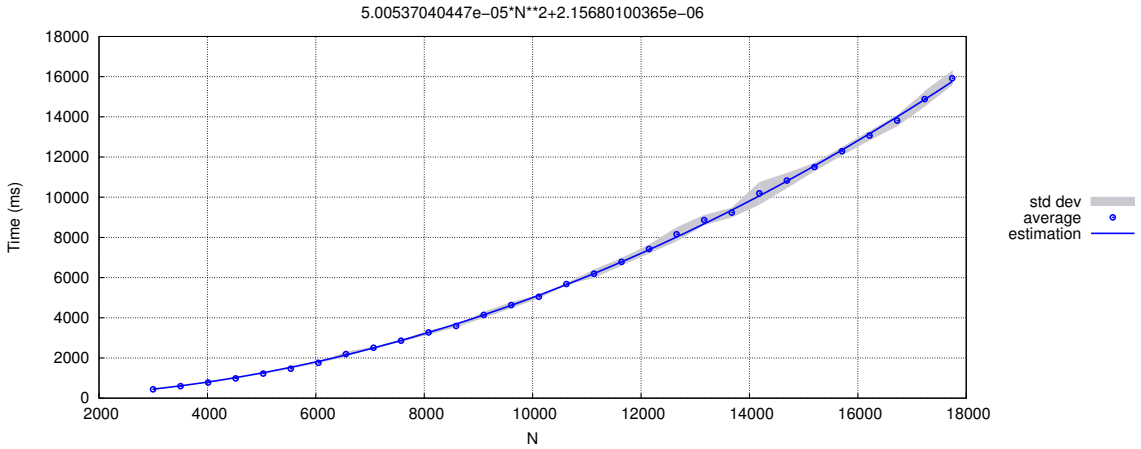


Figura 34 – Gráfico de execução do InsertionSort em Python em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.



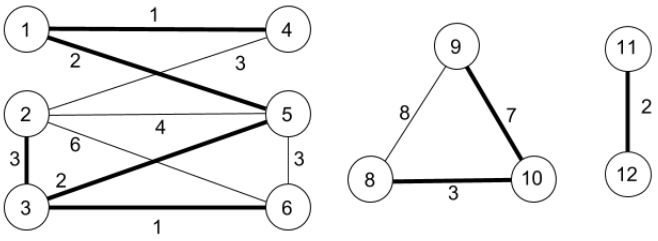
4.3.5 Floresta geradora mínima por Kruskal

Um grafo G é *conexo* se existe caminho conectando cada par de vértices de G. Um grafo G é *acíclico* se G não possuir ciclos. Uma *árvore* é um grafo acíclico e conexo. Uma *árvore geradora* de um grafo G é uma árvore F tal que $V(F) = V(G)$ e $E(F) \subseteq E(G)$. Seja G um grafo com peso $w(e)$ em cada aresta $e \in E(G)$. A *árvore geradora mínima* de G é uma árvore geradora F de G de menor peso $w(F)$ dentre todas as árvore geradoras de G, onde

$$w(F) = \sum_{e \in E(F)} w(e) \ .$$

Naturalmente, se G é um grafo desconexo, G não admite árvore geradora mínima. Definiremos então que uma *floresta geradora mínima* de G é uma floresta F de G tal que, para cada componente conexa C de G, vale que $F[C]$ é uma árvore geradora mínima de $G[C]$. A Figura 35 ilustra uma floresta geradora mínima de um dado grafo. As arestas em negrito são aquelas pertencentes à floresta geradora mínima.

Figura 35 – Exemplo de um grafo e uma floresta geradora mínima deste grafo.



O problema de determinar uma floresta geradora mínima é uma extensão do problema clássico de encontrar uma árvore geradora mínima. A extensão está em permitir que o grafo de entrada seja desconexo. Um algoritmo clássico para o problema de árvores geradoras mínimas é o algoritmo de Kruskal, de complexidade $O(m \log n)$. O algoritmo é diretamente aplicado para obter florestas geradoras mínimas, sem alterações (Algoritmo 15.

Para se chegar a complexidade de $O(m \log n)$, é necessário que o uso da estrutura de dados *união disjunta*, que implementa a união de conjuntos disjuntos e o teste se dois elementos estão no mesmo conjunto em tempo $O(\log n)$ [5].

Algoritmo 15 Algoritmo de Kruskal

Entrada: Grafo G
Saída: Uma floresta geradora mínima de G

```
1: função KRUSKAL(G)
2:     F ← (V(G), ∅)
3:     E ← ORDENAR(E(G))                    ▷ ascendentemente por peso
4:     para cada e ∈ E faça
5:         se "E(F) ∪ {e} é acíclico" então
6:             E(F) ← E(F) ∪ {e}
7:     retornar F
```

Como o EMA reporta a complexidade assintótica apenas de uma variável por análise, neste experimento aplicamos o algoritmo de Kruskal a grafos que possuem apenas 10% do número máximo de arestas. Essa escolha foi feita devido a limitação da memória da máquina para números muito grandes de arestas. Sabemos que o número máximo de arestas de um grafo é dado por $m = n(n-1)/2$. Fazendo a substituição, temos que a complexidade do algoritmo de Kruskal para um grafo que possui apenas 10% do número máximo de arestas é a mesma que a de um grafo completo, logo, $\Theta(n^2 \log n)$. Para a implementação em C, o teste do *Big-Enough* reportou $N_{\text{be}} = 23\,456$ e $T_{\text{be}} = 17\,372$ ms. Para a implementação em Python, o teste reportou $N_{\text{be}} = 4\,484$ e $T_{\text{be}} = 20\,831$ ms. As Figuras 36 e 37 ilustram a execução do algoritmo de Kruskal em C e Python através do EMA para o teste do *Big-Enough*.

Figura 36 – Gráfico de execução do Kruskal em C para grafos que possuem 10% do número máximo de arestas em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.

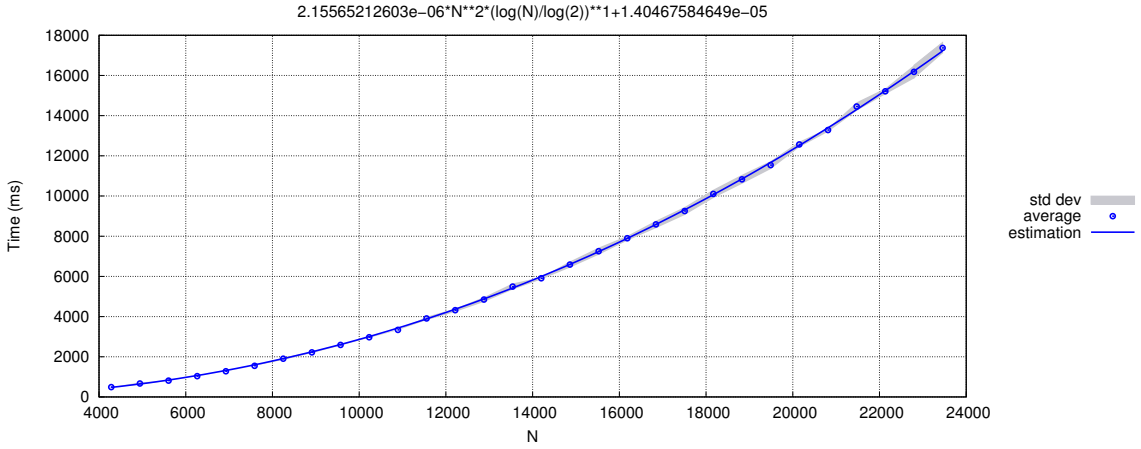
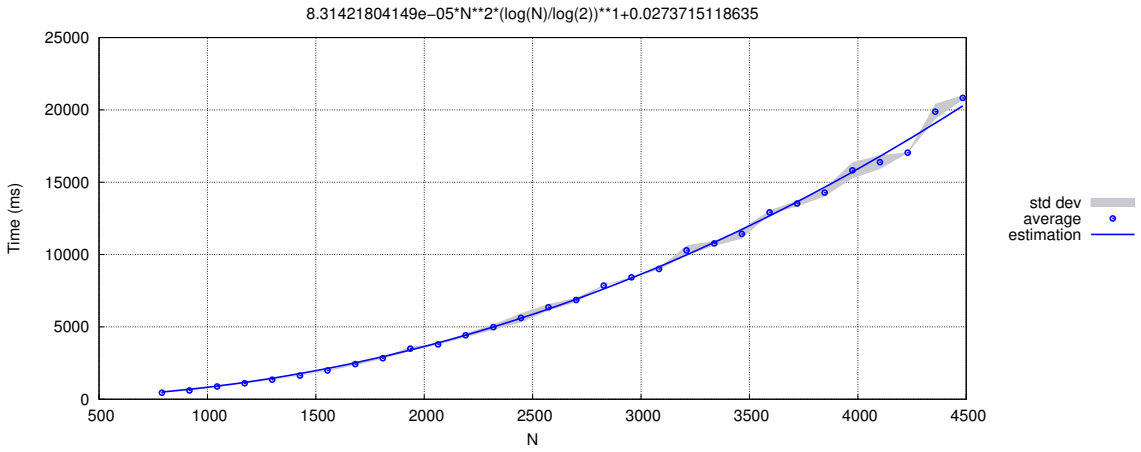


Figura 37 – Gráfico de execução do Kruskal em Python para grafos que possuem 10% do número máximo de arestas em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.



4.3.6 StoogeSort

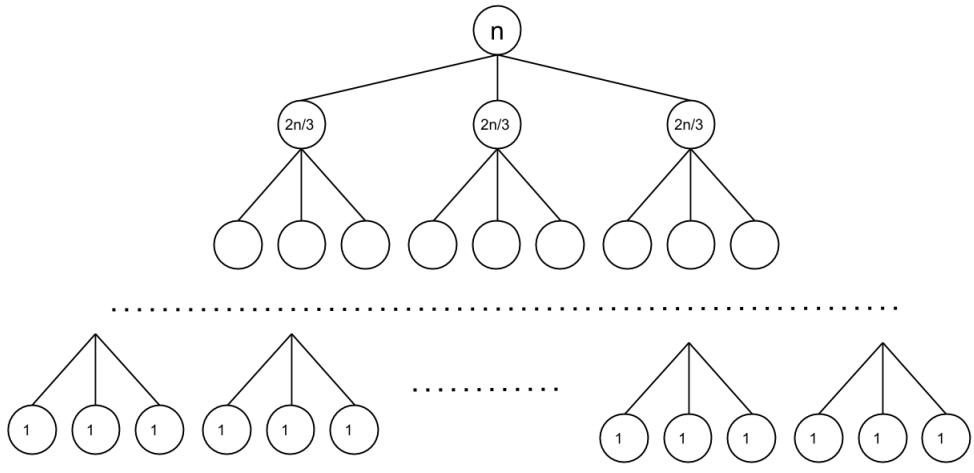
O *StoogeSort* é um algoritmo de ordenação que utiliza a técnica de divisão e conquista, porém, pouco usado por ter o pior desempenho se comparado aos outros algoritmos de ordenação. O StoogeSort possui complexidade $O(n^{2.71})$ tanto no pior quanto no melhor caso, pois independe da ordem dos elementos da lista. A ideia deste algoritmo é definida da seguinte maneira. Se o primeiro elemento da lista é maior que o último, trocam-se os dois. Se na lista há três ou mais elementos, realizam-se três chamadas recursivas consecutivas. Uma para 2/3 dos valores iniciais da lista, uma para 2/3 dos valores finais da lista e novamente uma para 2/3 dos valores iniciais da lista. O StoogeSort está descrito no Algoritmo 16.

Algoritmo 16 StoogeSort

```
1: função STOOGESORT(L,i,j)
2:     se L[i] > L[j] então
3:         L[i],L[j] ← L[j],L[i]
4:     se (j − i + 1) > 2 então
5:         t ← (j − i + 1)/3
6:         STOOGESORT(L, i, j − t)
7:         STOOGESORT(L, i + t, j)
8:         STOOGESORT(L, i, j − t)
9:     retornar L
```

Como o StoogeSort é um algoritmo de divisão e conquista, cuja complexidade da fase de conquista é constante, a preocupação sobre a escolha das entradas descrita no início do capítulo se faz necessária. A escolha de pontos para o experimento foi feita da seguinte maneira. Considere a árvore de recursão ilustrada na Figura 38. Seja n o tamanho da lista a ser ordenada. Note que cada nó representa a diminuição do número de elementos a ser ordenado para 2/3 em relação ao número de elementos originais até que só existam listas com 1 ou 2 elementos representadas pelas folhas. Logo, o algoritmo pára quando $\lceil (2n/3)^h \rceil \leq 2$, onde h é a altura da árvore de recursão. Note que se $n = 1$ não é feita nenhuma recursão. Para $n = \lfloor 3/2 \rfloor$, temos $h = 2$. Para $n = \lfloor (3/2)^2 \rfloor$, temos $h = 3$, e assim sucessivamente. Logo, os valores de n escolhidos para incrementar a altura da árvore a cada nova entrada são potências de 3/2. Para fazer com que o teste do *Big-Enough* escolhesse tais pontos, foi utilizada uma função ARR diferente da identidade: dado um valor de N, ela retorna a potência de $\lfloor 3/2 \rfloor$ mais próxima de tal N.

Figura 38 – Árvore de recursão do StoogeSort.



Para a implementação em C, o teste do *Big-Enough* reportou $N_{\text{be}} = 8\,091$ e $T_{\text{be}} = 54\,859$ ms. Para a implementação em Python, o teste reportou $N_{\text{be}} = 1\,065$ e $T_{\text{be}} = 7\,572$ ms. Porém, para esta versão do algoritmo o EMA não detectou a função correta utilizando o limiar de equivalência $le = 0.5\%$. Para isso, foi necessário utilizar um valor de le contido no intervalo [0.069%,0.267%). As Figuras 39 e 40 ilustram a execução do StoogeSort em C e Python através do EMA para o teste do *Big-Enough*.

Figura 39 – Gráfico de execução do StoogeSort em C em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.

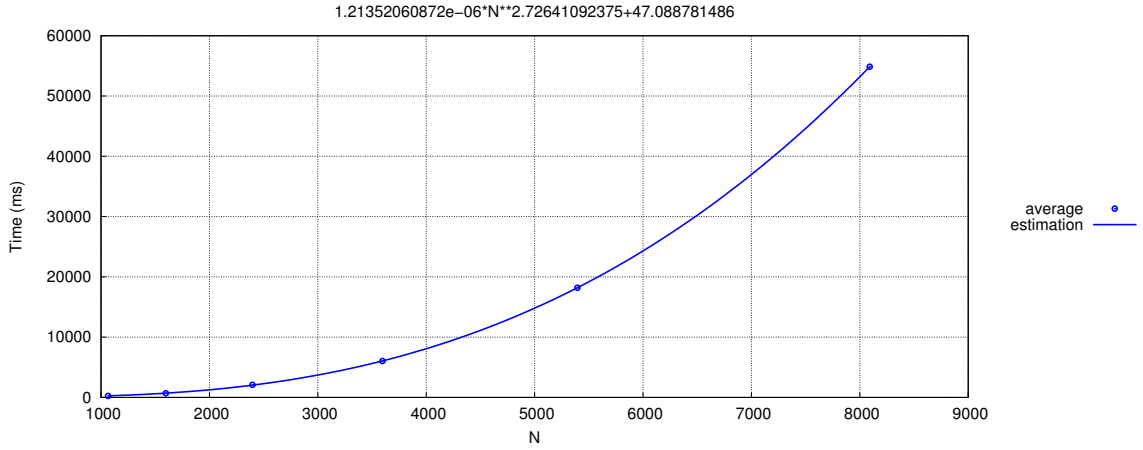
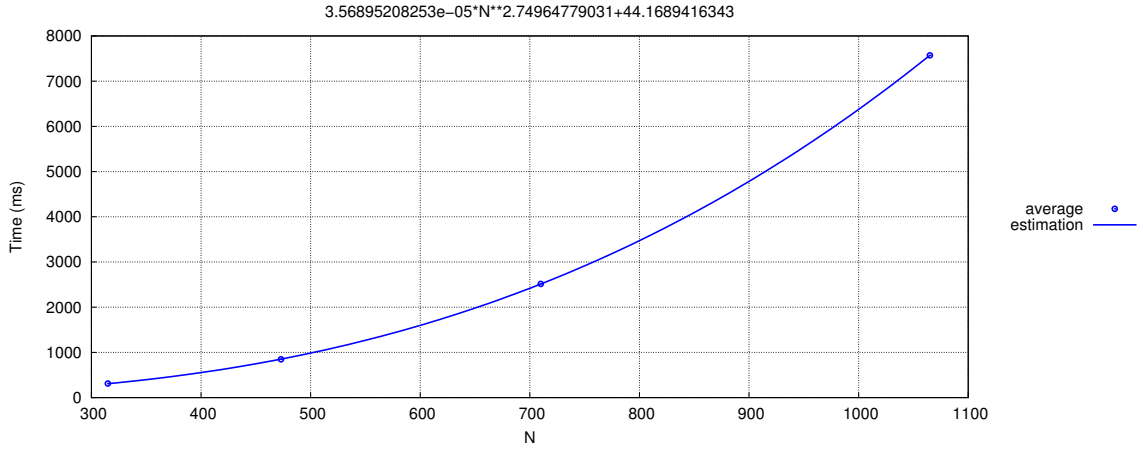


Figura 40 – Gráfico de execução do StoogeSort em Python em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.



4.3.7 Multiplicação de matrizes pelo método de Strassen

O algoritmo de multiplicação de matrizes tradicional implementa a ideia direta a partir da definição que consiste uma multiplicação de matrizes, a saber

$$R = A \times B \Leftrightarrow R[i,j] = \sum_{k=1}^{N} A[i,k] \times B[k,j], \text{ para todo } 1 \leq i,j \leq N,$$

onde supõe-se que R, A, B são matrizes quadradas de dimensão N (a definição de multiplicação de matrizes é mais geral que a fornecida, mas esta é suficiente para nossos propósitos). A complexidade do algoritmo que decorre desta definição é $\Theta(N^3)$.

O algoritmo de multiplicação de matrizes publicado de Strassen [47] é assintoticamente menor que a complexidade do método tradicional. Foi o primeiro algoritmo que provou que o algoritmo decorrente da definição de multiplicação de matrizes não é ótima e tem um interessante histórico por baixas sucessivas em sua complexidade, onde a cada novo algoritmo, a melhora se dava em termos de décimos ou centésimos no expoente do polinômio [48].

O algoritmo de Strassen utiliza a técnica de divisão e conquista da seguinte forma. Primeiro, particiona-se as matrizes A e B em quatro submatrizes de dimensão $N/2 \times N/2$ como esquematizado em (4.3). Em seguida, determina-se os valores p_i para todo $1 \leq i \leq 7$ tal que a multiplicação matricial envolvida no cálculo de cada p_i é determinada recursivamente (fase de divisão). A ideia do método consiste na observação de que, a partir destes valores, é possível obter a matriz $R = A \times B$ conforme o esquema (fase de conquista).

$$\begin{matrix} \\ A \end{matrix}\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}\begin{matrix} \\ B \end{matrix}\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{matrix} \\ R \end{matrix}\begin{bmatrix} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{bmatrix}, \tag{4.3}$$

onde:

$$\begin{array}{lll} p_1 = A_{11}(B_{12} - B_{22}); & p_2 = (A_{11} + A_{12})B_{22}; & p_3 = (A_{21} + A_{22})B_{11}; \\ p_4 = A_{22}(B_{21} - B_{11}); & p_5 = (A_{11} + A_{22})(B_{11} + B_{22}); & \\ p_6 = (A_{12} - A_{22})(B_{21} + B_{22}); & p_7 = (A_{11} - A_{21})(B_{11} + B_{12}). & \end{array}$$

Deste modo, são feitas 7 multiplicações de matrizes de dimensão $N/2 \times N/2$. A soma de duas matrizes $N \times N$ é efetuada em tempo $\Theta(N^2)$. Assim, se $T(N)$ corresponde a complexidade do algoritmo para multiplicação de matrizes $N \times N$, $T(N)$ pode ser descrita pela equação de recorrência $T(N) = 7T(N/2) + \Theta(N^2)$ se $N > 1$, e $T(1) = \Theta(1)$, cuja resolução resulta em $T(N) = \Theta(N^{\log_2 7}) \approx \Theta(N^{2.81})$. O Strassen está descrito no Algoritmo 17. Como o algoritmo de Strassen é um algoritmo de divisão-e-conquista, apesar do efeito degrau ser suavizado pela complexidade não-constante da fase de conquista, adotamos a geração das entradas utilizando o mesmo raciocínio da busca binária e do StoogeSort, conforme visto na Seção 4.3 e 4.3.6, respectivamente.

Figura 41 – Gráfico de execução do Strassen em C em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.

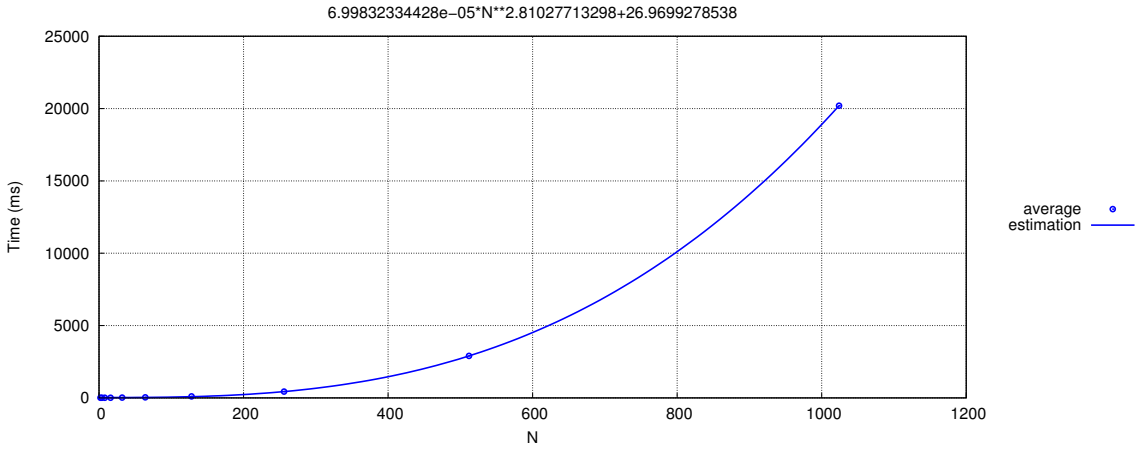
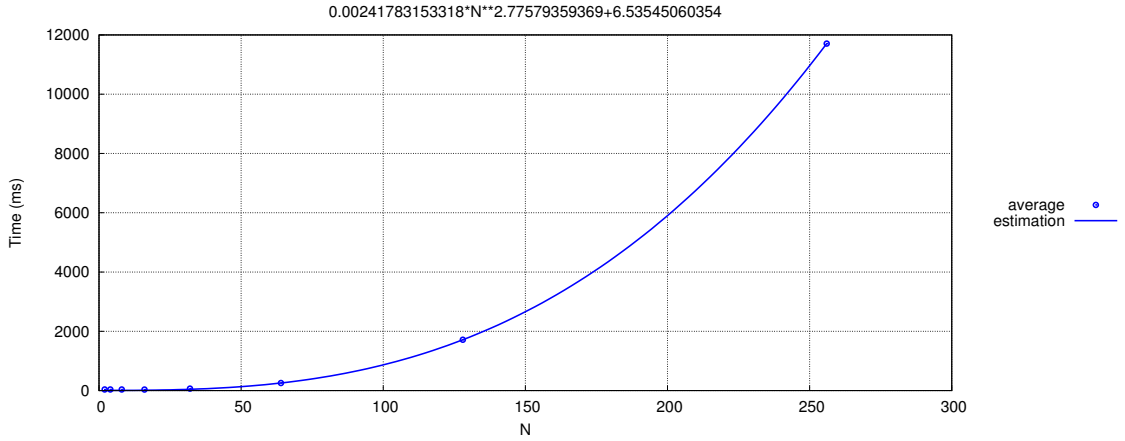


Figura 42 – Gráfico de execução do Strassen em Python em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.



Algoritmo 17 Strassen

função STRASSEN(A, B, N)
  se $N = 1$ **então**
    $R_{11} \leftarrow A_{11} \times B_{11}$
  senão
    Sejam $S1$,$S2 \ldots S10$ matrizes $N/2 \times N/2$.
    Sejam $P1$,$P2 \ldots P7$ matrizes $N/2 \times N/2$.
    ▷ Calculando a soma/subtraçao das submatrizes.
    $S1 \leftarrow B_{12} - B_{22}$
    $S2 \leftarrow A_{11} + A_{12}$
    $S3 \leftarrow A_{21} + A_{22}$
    $S4 \leftarrow B_{21} - B_{11}$
    $S5 \leftarrow A_{11} + A_{22}$
    $S6 \leftarrow B_{11} + B_{22}$
    $S7 \leftarrow A_{12} - A_{22}$
    $S8 \leftarrow B_{21} + B_{22}$
    $S9 \leftarrow B_{21} + B_{22}$
    $S10 \leftarrow A_{11} - A_{21}$
    ▷ Calculando o produto de matrizes.
    $P1 \leftarrow$ STRASSEN$(A_{11}, S1)$
    $P2 \leftarrow$ STRASSEN$(S2, B_{22})$
    $P3 \leftarrow$ STRASSEN$(S3, B_{11})$
    $P4 \leftarrow$ STRASSEN$(A_{22}, S4)$
    $P5 \leftarrow$ STRASSEN$(S5, S6)$
    $P6 \leftarrow$ STRASSEN$(S7, S8)$
    $P7 \leftarrow$ STRASSEN$(S9, S10)$
    ▷ Calculando o resultado final da multiplicação de $A \times B$.
    $R_{11} \leftarrow P5 + P4 - P2 + P6$
    $R_{12} \leftarrow P1 + P2$
    $R_{21} \leftarrow P3 + P4$
    $R_{22} \leftarrow P1 + P5 - P3 - P7$
  retornar R

Para a implementação em C, o teste do *Big-Enough* reportou $N_{\text{be}} = 1\,024$ e $T_{\text{be}} = 20\,200$ ms. Para a implementação em Python, o teste reportou $N_{\text{be}} = 256$ e $T_{\text{be}} = 11\,707$ ms. As Figuras 41 e 42 ilustram a execução do algoritmo de Strassen através do teste do EMA para o *Big-Enough*. É possível observar que a complexidade encontrada pelo EMA, em ambas implementações, é muito próxima da complexidade teórica.

4.3.8 Torre de Hanói

O problema *Torre de Hanói* é um quebra-cabeça que tem por objetivo transferir uma certa quantidade de discos dispostos em uma haste A, ordenados por diâmetro (o de maior diâmetro mais abaixo na pilha), para uma haste B utilizando uma haste C como auxiliar. Iterativamente, escolhe-se um disco de qualquer haste que esteja por cima e o insere em outra haste a escolha, desde que tal disco seja o de menor diâmetro daquela haste escolhida. O algoritmo clássico para resolver este problema utiliza a abordagem de divisão e conquista, cuja complexidade de tempo $\Theta(2^n)$ é sabida ser ótima.

Algoritmo 18 Torre de Hanói

```
1: função HANOI(n,a,b,c)
2:     se n > 0 então
3:         x ← HANOI(n-1, a,c,b)
4:         escrever (a," → ",b)
5:         x ← x+HANOI(n − 1, c,b,a)
6:     retornar x
```

Para a implementação em C, o teste reportou $N_{\text{be}} = 35$ e $T_{\text{be}} = 93\,724$ ms. Para a implementação em Python, o teste reportou $N_{\text{be}} = 29$ e $T_{\text{be}} = 116\,162$ ms. As Figuras 43 e 44 ilustram a execução do algoritmo através do EMA para o teste do *Big-Enough*.

Figura 43 – Gráfico de execução de Hanói em C em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.

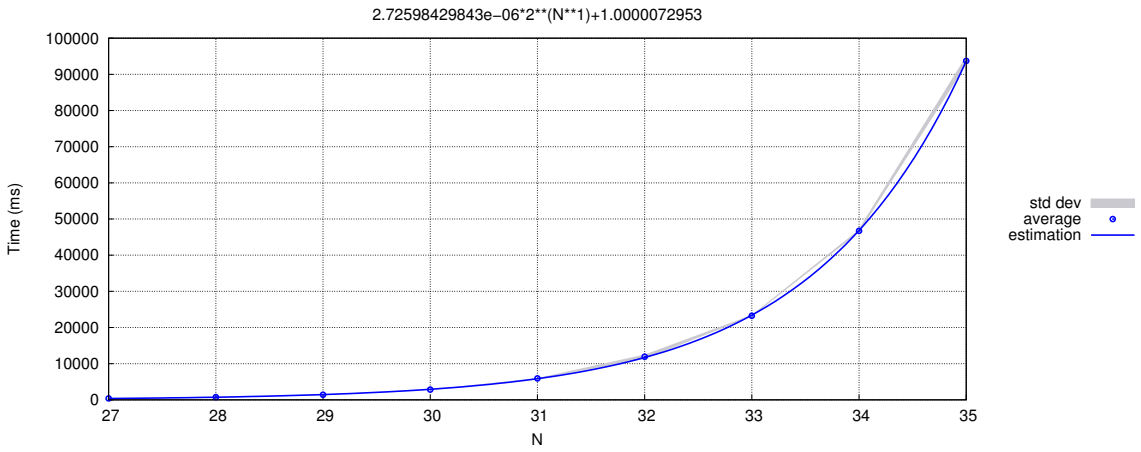
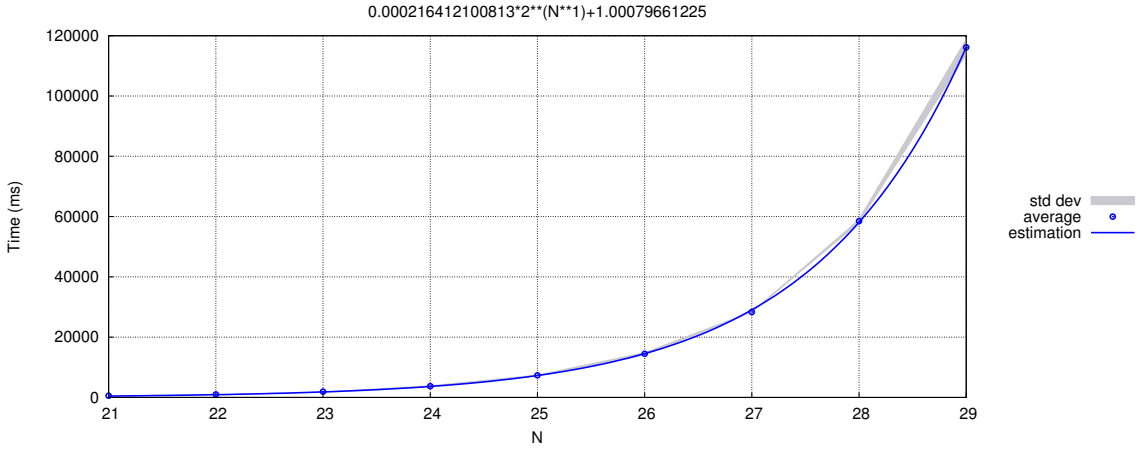


Figura 44 – Gráfico de execução de Hanói em Python em que os valores da entrada foram gerados durante a aplicação do teste do *Big-Enough*.



4.4 Análise dos resultados

A Tabela 11 resume os resultados encontrados nos experimentos realizados com os algoritmos reais e sintéticos implementados em C e Python. Uma primeira análise a ser feita é a comparação entre os algoritmos reais e suas respectivas funções sintéticas. Sabemos que a constante multiplicativa do algoritmo real é maior que aquela do sintético, já que o algoritmo real, em geral, executa mais instruções do que o sintético que executa apenas um incremento dentro de um laço de repetição. Sendo assim, temos que a expectativa é que $N_{\text{inf-R}} < N_{\text{inf-S}}$ e $dN_R < dN_S$ (todos os sufixos S e R nas variáveis correspondem a estes parâmetros para o algoritmo sintético e real, respectivamente). Seja N_i o i-ésimo valor de N testado pelo *Big-Enough*. Como consequência, temos que $N_{\text{1-R}} < N_{\text{1-S}}$. Se ambos foram encontrados na primeira iteração, então $N_{\text{be-R}} = N_{\text{1-R}} < N_{\text{1-S}} = N_{\text{be-S}}$, e logo, $N_{\text{be-R}} < N_{\text{be-S}}$ e $T_{\text{be-S}} < T_{\text{be-R}}$. Por contrapositiva, se $N_{\text{be-R}} \geq N_{\text{be-S}}$, então eles não foram provavelmente encontrados na primeira iteração. Pela Tabela 11, observamos que essas relações são válidas para todos os experimentos (inclusive para aqueles em que essas relações não são estritamente válidas, pelo fato de terem valores muito próximos). Além disso, é possível notar que podemos utilizar funções sintéticas para prever os valores de T_{be} de algoritmos reais, pois esses valores estão na mesma ordem de grandeza (os valores de N_{be} são distintos pois, como observamos, as constantes multiplicativas entre algoritmo sintético e real são diferentes).

A segunda análise a ser feita é a comparação entre os algoritmos reais em C e Python. De acordo com as desigualdades (4.1) e (4.2), observando a Tabela 11, podemos notar que essas relações são válidas para a maioria dos experimentos (inclusive para aqueles em que essas relações não são estritamente válidas, pelo fato de terem valores muito próximos). Exceto para o Strassen, que na sua versão real em C precisou de mais iterações para ser detectada do que em sua versão real em Python, por isso o valor de T_{be} do programa real C é maior que o T_{be} do programa em Python. Além disso, é possível observar que os valores de T_{be} em todas as colunas e cada linha possuem a mesma ordem de grandeza. Os gráficos de todas as execuções dos algoritmos reais e das funções sintéticas estão disponíveis em uma página [49] para eventuais consultas.

Tabela 11 – Comparação entre os programas reais e sintéticos.

Algoritmo	Complexidade	Programa real em C			Programa real em Python			Função sintética em C			Função sintética em Python		
		N_{be}	T_{be}	*it*	N_{be}	T_{be}	*it*	N_{be}	T_{be}	*it*	N_{be}	T_{be}	*it*
Mediana das Medianas	$\Theta(n)$	26 653 266	3 922	1	1 029 231	4 954	1	2 512 695 342	3 741	1	48 095 705	3 977	1
MergeSort	$\Theta(n \log n)$	16 441 378	4 559	1	1 198 744	4988	2	95 748 865	4 352	1	2 881 606	5 092	2
MergeSort long	$\Theta(n \log^2 n)$	4 661 611	5 341	1	393 088	6 802	2	5 661 923	4 881	1	219 340	5 659	1
InsertionSort	$\Theta(n^2)$	93 287	14 479	1	17 742	15 919	2	94 937	15 427	1	13 801	15 718	1
Kruskal	$\Theta(n^2 \log n)$	23 456	17 372	1	4 484	20 831	2	25 495	16 272	1	4 034	16 064	1
StoogeSort	$O(n^{2.7})$	8 091	54 859	3	1 065*	7 572	1	8 091	47 292	2	1 065**	8 779	1
Strassen	$\Theta(N^{2.8})$	1 024	20 200	7	256	11 707	5	4 096	16 173	9	1 024	16 118	7
Hanói	$\Theta(2^n)$	35	93 724	1	29	116 162	1	36	95 187	1	30	88 214	1

*Valor encontrado utilizando limiar de equivalência (le) contido no intervalo [0.00069,0.00267)
**Valor encontrado utilizando limiar de equivalência (le) contido no intervalo [0.00008,00038)

4.5 Considerações finais

A análise de algoritmos através da abordagem analítica fornece a complexidade do recurso medido por meio de uma função assintótica, ou seja, em função de seu termo de maior crescimento. A função precisa que determina a complexidade do algoritmo pode conter monômios com constante multiplicativa muito grande em relação ao monômio de maior crescimento. Para valores pequenos de entrada N, tais monômios dominam a função, que terá valores aproximados por tais monômios. Contudo, conforme N cresce e torna-se suficientemente grande, os termos de menor crescimento podem ser desprezados. A abordagem analítica não fornece o valor inicial de tamanho de entrada para o qual o monômio de maior crescimento domina a função. A metodologia do teste do *Big-Enough* é uma forma de se obter tal valor inicial, que pode ser aplicada a qualquer método empírico de se obter a complexidade. Neste capítulo, aplicamos esta metodologia ao EMA utilizando algoritmos reais para representar as classes mais comuns de complexidade conhecidos na literatura e um programa sintético.

A partir de um programa sintético $\mathcal{A}$ tal que a sua complexidade de tempo é dada pela função $f(N,a,b,c,d) = a^{N^b} \cdot N^c \cdot (\log_2 N)^d + 200$, é possível controlar a complexidade do algoritmo através dos parâmetros a,b,c,d. O teste do *Big-Enough* foi importante neste experimento, pois determinou o valor inicial de tamanho de entrada N para o qual verifica-se empiricamente a complexidade analítica para cada tipo de função, conforme os valores dos parâmetros a,b,c,d. Observou-se que para funções exponenciais não precisa-se crescer muito o valor de N para que seu comportamento assintótico seja determinado corretamente. Em contrapartida, as funções polilogarítmicas crescem muito lentamente e detectar este tipo de complexidade corretamente é necessário valores de N muito maiores do que para funções exponenciais e polinomiais.

Ao comparar os algoritmos reais e sintéticos em C com seus equivalentes em Python, observamos que para detectar corretamente suas complexidades, os programas escritos em C encontraram valores de N maiores do que aqueles implementados em Python. Por outro lado, os tempos de execução de ambas implementações estão muito próximos, na mesma ordem de grandeza. Isso ocorre pois a linguagem C é mais eficiente que Python e as constantes multiplicativas da função tempo associadas a C são menores que aquelas associadas aos algoritmos equivalentes em Python.

Ao comparar os algoritmos reais com seus equivalentes sintéticos de mesma linguagem, vimos que os valores de N para os algoritmos reais são menores que aqueles encontrados para os sintéticos desde que sejam encontrados na mesma iteração. Isso ocorre porque a constante multiplicativa da função do algoritmo real é maior que a do algoritmo sintético pois os algoritmos reais executam um maior número de instruções. Apesar disso, seus tempos de execução estão na mesma ordem de grandeza.

Este experimento foi importante pois observamos que é possível utilizar programas sintéticos para prever os tamanho de entrada para o qual é possível detectar complexidade de algoritmos reais. Além disso, apesar de ser dito na literatura que a complexidade de um algoritmo é dada por uma função $T(n)$ para valores de n suficientemente grandes, o teste do *Big-Enough* mostrou que, para os algoritmos reais objetos de estudo abrangendo diversas classes de complexidade, esses valores não precisam ser muito grandes para que a função se caracterize como tal.

CONCLUSÃO

A proposta deste trabalho é estudar a análise automatizada de algoritmos. Em especial, aquela conduzida pela via empírica é de muita importância nas mais diversas situações, tais como: (i) o uso didático para se constatar uma complexidade teórica; (ii) para conferir uma implementação de um algoritmo, certificando que ela foi feita conforme o projetado; (iii) quando não se tem acesso ao código-fonte (por exemplo, no caso de bibliotecas externas e programas de terceiros); (iv) aferir as constantes multiplicativas de algoritmos que possuem a mesma complexidade analítica; (v) prever a quantidade de tempo/memória necessários para execução do algoritmo para uma entrada desconhecida; (vi) testar o algoritmo em instâncias reais, ao invés de por análise de pior caso, entre outras aplicações.

Desde a década de 70, foram realizadas diversas pesquisas com o objetivo de desenvolver ferramentas que realizam análise automatizada de algoritmos. Conforme abordado no Capítulo 1, há aquelas que fazem a análise através do método analítico, na qual é feita uma análise do código, tais como ACE [26], ACME [28], ANAC [29], RAML [45]. Em contrapartida, existem as ferramentas que realizam análise empírica tais como METRIC [25], Trend Profiler [30], Aprof [33], AlgoProf [34], MOCCA [35] e EMA [1]. A maior parte dessas ferramentas possuem muitas limitações e fazem apenas análise de determinados tipos de algoritmos tais como linguagem de programação específica, apenas um paradigma de programação e somente um tipo de função de complexidade. Dentre todos, o EMA diferencia-se por realizar estimativas de complexidade assintótica de algoritmos em diversas linguagens de programação, em paradigma funcional e imperativo, e aborda diversas classes de complexidade (polilogarítmica, polinomial e exponencial).

No Capítulo 2, apresentamos a metodologia e as funcionalidades do EMA. Esta ferramenta fornece uma estimativa da complexidade de tempo/espaço de algoritmos em ambientes heterogêneos através de várias execuções do mesmo para diversos tamanhos de entrada, monitorando o uso de recursos e armazenando seus consumos. São executadas diversas regressões não-lineares para determinar os parâmetros de funções que descrevem a complexidade do algoritmo, obtendo-se um conjunto de funções candidatas. Para escolher uma dentre elas, o EMA seleciona a função mais simples (conforme os critério abordados na Seção 2.5.3). Além disso, o EMA também gera automaticamente os tamanhos de entrada baseado em parâmetros determinados pelo usuário, tais como limites de tempo e espaço.

No Capítulo 3, o EMA foi comparado com as ferramentas Trend Profiler, Aprof e RAML. Como o Trend Profiler e o Aprof analisam apenas programas em C e o RAML apenas programas em OCaml, foram realizados estudos de caso separadamente. A comparação entre o EMA, Trend Profiler e Aprof foi feita em dois estudos de caso: BubbleSort e MergeSort. O EMA obteve os resultados mais precisos por fornecer uma função que descreve a complexidade do algoritmo sem a necessidade do usuário intervir no processo e conhecer previamente a complexidade dos algoritmos analisados. Por outro lado, o Aprof necessita que o usuário selecione funções em uma lista até que observe que o gráfico convergiu para uma constante não negativa. Essa é uma limitação importante do Aprof, pois

existem diversas funções que podem convergir para uma constante (vide conceito de funções equivalentes do EMA, abordado na Seção 2.5.3.3) e se a complexidade do algoritmo em análise for desconhecida, o usuário precisa (i) descobrir o conjunto de funções candidatas a serem a complexidade do algoritmo e (ii) decidir, a cada novo algoritmo, qual o critério para a escolha. A escolha de desempate do EMA (função melhor-palpite, conforme Seção 2.5.3.4) é um critério único escolhido para funcionar em uma vasta aplicação de análises. Além disso, o Aprof em seu estado atual não possui funções exponenciais em sua lista, logo, não é possível analisar corretamente algoritmos desta natureza. O Trend Profiler também mostrou limitações importantes, tais como perfazer regressão linear em escala logarítmica. A minimização do erro da função nesta escala não corresponde precisamente à minimização de erro em escala normal e, por isso, a qualidade inferior do resultado final comparado às outras ferramentas é evidente. Além disso, o Trend Profiler fornece como funções alternativas apenas polinômios de expoentes inteiros na vizinhança do expoente do polinômio encontrado pela regressão, porém aqueles com grau limitado a 3. Ele apresenta os gráficos e fica totalmente a critério do usuário definir qual é a função de complexidade do algoritmo. Assim, o usuário fica diante do mesmo problema que ocorre com o Aprof. O Trend Profiler limita-se a análise de algoritmos da classe polinomial e fornece um gráfico de residuais para que o usuário possa refinar sua análise, como por exemplo, tentar deduzir a partir do gráfico de residuais se há um termo multiplicativo logarítmico. Porém, apenas de posse de tal gráfico, não há método específico para determinar se este de fato é o caso ou, mesmo em caso afirmativo, determinar o expoente desse logaritmo.

A comparação com o RAML foi feita através de dois estudos de caso. O primeiro foi um experimento para análise de pior caso do QuickSort. Inicialmente implementado utilizando recursão comum, não foi possível crescer o suficiente o tamanho da entrada para execução no EMA devido ao estouro de pilha. Como solução, foi implementado tal algoritmo com recursão de cauda. O mesmo código foi analisado pelo EMA e pelo RAML e os resultados foram, respectivamente $\Theta(N^2)$ e $O(N^3)$. O EMA encontrou de maneira precisa o resultado da complexidade analítica que é $\Theta(N^2)$ ao passo que o RAML encontrou um limite superior não-justo. Apesar disso, para a versão do QuickSort que não utiliza recursão de cauda, o RAML encontrou o limite justo $O(N^2)$. Este estudo torna evidente que as ferramentas que utilizam a via da análise de código podem ser susceptíveis à forma que o algoritmo encontra-se descrito, mesmo que equivalentes do ponto de vista de complexidade. No segundo estudo de caso, foi feita uma análise de pior caso do algoritmo busca em profundidade, que possui complexidade $\Theta(n + m)$, onde n e m correspondem, respectivamente, ao número de vértices e arestas de um dado grafo de entrada. Para tal experimento no EMA, foi fixada uma variável em uma constante e variou-se a outra e obtivemos $\Theta(m)$ (para n fixo) e $\Theta(n)$ (para m fixo), portanto, complexidades coerentes com aquela obtida pela abordagem analítica. O RAML não foi capaz de analisar este algoritmo. Em contato com o responsável pelo projeto do RAML, ele disse que uma possível razão é que o RAML teve dificuldade em "concluir" que a busca em profundidade pára. Apesar desta conclusão ser trivial para a análise humana (com a ideia de que cada vértice não é visitado mais de uma vez devido às marcações de vértices e arestas à medida que a busca progride), ela representa, conforme demonstrado neste estudo de caso, que pode ser uma barreira para análises automatizadas de código.

O método analítico clássico de se obter a complexidade a partir da descrição do algoritmo não determina a constante multiplicativa presente implicitamente na notação O e família, mas que as funções obtidas governam o consumo de recursos para entradas "grandes o suficiente". No Capítulo 4, propomos o teste do *Big-Enough*, que tem

por objetivo determinar o valor inicial de tamanho de entrada N para o qual, com uma execução de entradas de tamanhos variados limitados por N, verifica-se empiricamente a complexidade analítica com uma dada ferramenta empírica. Foram realizados experimentos com a ferramenta EMA para funções sintéticas e algoritmos reais implementados em C e Python. Nos testes com as funções sintéticas, observou-se que para a classe de funções polilogarítmicas foram encontrados os maiores valores de N como resultado do teste, pois funções logarítmicas crescem muito lentamente. Em contrapartida, a função exponencial por crescer rapidamente, com um baixo valor de N foi possível detectar seu comportamento. Nos testes realizados com algoritmos reais, observou-se que seus equivalentes sintéticos na mesma linguagem de programação possuem valores de N maiores do que os reais, desde que encontrados na mesma iteração. Isso se explica pelo fato de algoritmos reais executarem um maior número de instruções do que a função sintética que produzimos. Por isso, a constante multiplicativa da função de complexidade associadas aos algoritmos reais foram maiores que aquela da função sintética. Contudo, os tempos de execução associado a tais tamanhos de entrada "grandes o suficiente" estão na mesma ordem de grandeza. Na comparação entre os algoritmos em C e Python, observou-se que algoritmos implementados em C necessitam de valores de N maiores para encontrar a complexidade correta. Isso ocorre porque a linguagem C é mais eficiente que Python por consumir menos memória e menos processamento. Consequentemente, as constantes multiplicativas da função tempo associadas a C são menores que aquelas relativas as versões equivalentes em Python. Esses experimentos mostraram que algoritmos reais e sintéticos precisam de tempos de execução "grandes o suficiente" muito próximos para serem detectados. Logo, este estudo parece indicar que é possível estender as conclusões obtidas por funções artificiais para algoritmos reais, dado que os algoritmos foram escolhidos arbitrariamente, com o único critério de cobrir razoavelmente o espectro de classes de complexidade. Além disso, observou-se que não é necessário que os valores de N cresçam muito para que sua função de complexidade seja detectável empiricamente. Nos estudos realizados, foi visto que independente da função em análise, o tempo grande o suficiente para que tal abordagem empírica tenha êxito não chega, em geral, a 1 minuto para funções polinomiais e polilogarítmicas, sendo a maior parte deles em torno de 45 segundos e em torno de 1,5 minuto para funções exponenciais. Os experimentos realizados com algoritmos de diversas classes de complexidade contribuíram com melhorias na metodologia do EMA, tornando possível obter corretamente funções que na versão anterior a este trabalho não eram.

Como questões de interesse em aberto, propomos as seguintes:

- aplicar o teste Big-Enough a outras ferramentas de análise de algoritmos, e compará-las ao EMA nestes termos;

- estender o teste Big-Enough a outros algoritmos de classes de complexidade não abordadas neste trabalho, na tentativa de fortalecer ou confrontar as conclusões aqui obtidas;

- propor uma metodologia para o EMA com o objetivo de análise de algoritmos em mais de uma variável. Note que, conforme resultados apresentados no Capítulo 3 no qual analisamos algoritmos em grafos, o EMA conduz apenas análises de complexidade parciais (uma variável por vez, fixando o valor de todas as outras). Assim, em busca de profundidade por exemplo, o EMA detectou a complexidade de $\Theta(m)$ e $\Theta(n)$ em tais análises parciais. A metodologia de interesse a que nos referimos deveria determinar se trata-se de um algoritmo $\Theta(nm)$, $\Theta(n+m)$ ou, ainda, de

$\Theta(m \log n + n \log m)$. Cabe ressaltar que, conforme [9], podem existir um número arbitrário de monômios não-desprezíveis na expressão de complexidade assintótica de algoritmos em mais de uma variável.

REFERÊNCIAS

[1] OLIVEIRA, F. S. *EMA - WebPage*. 2017. Acesso em 20 de Março de 2017. Disponível em: <http://fabianooliveira.ime.uerj.br/ema>.

[2] KNUTH, D. E. *The Art of Computer Programming*. 3.a. ed. Boston, USA: Addison-Wesley Professional, 1998. v. 1.

[3] SEDGEWICK, R.; FLAJOLET, P. *An Introduction to the Analysis of Algorithms*. 2.a. ed. Boston, USA: Addison-Wesley Professional, 2013.

[4] SEDGEWICK, R. *Coursera Videoaula - Analysis of Algorithms*. 2018. Acesso em 03 de Abril de 2018. Disponível em: <http://www.coursera.org/learn/analysis-of-algorithms>.

[5] CORMEN, T. H. et al. *Introduction to Algorithms*. 3.a. ed. London, England: The MIT Press, 2009.

[6] SZWARCFITER, J. L.; MARKENZON, L. *Estruturas de Dados e Seus Algoritmos*. 3.a. ed. Rio de Janeiro, Brasil: LTC, 2010.

[7] SPINRAD, J. *Efficient Graph Representations.: The Fields Institute for Research in Mathematical Sciences*. Providence, USA: American Mathematical Society, 2003.

[8] THE OXFORD MATH CENTER. *Analysis of Algorithms*. Acesso em 2 de Janeiro de 2018. Disponível em: <http://www.oxfordmathcenter.com/drupal7/node/639>.

[9] OLIVEIRA, F. S.; BARBOSA, V. C. A note on counting independent terms in asymptotic expressions of computational complexity. *Optimization Letters*, v. 11, p. 1757–1765, 2017.

[10] KURTZ, S. A.; SIMON, J. The undecidability of the generalized collatz problem. In: Anais da 4.a International Conference on Theory and Applications of Models of Computation. *Theory and Applications of Models of Computation*. Berlin, Germany, 2007. p. 542–553.

[11] OLIVEIRA, T.; SILVA. Empirical verification of the $3x+1$ and related conjectures. In: *The Ultimate Challenge: The* $3x+1$ *problem*. Providence, USA: American Mathematical Society, 2010. p. 189–207.

[12] ZUDIO, A. et al. Análise de complexidade do método de eliminação gaussiana em GPU através da ferramenta EMA. In: Anais do 17.o Simpósio em Sistemas Computacionais de Alto Desempenho. Aracaju, Brasil, 2016.

[13] SOUZA, R. et al. Empirical analysis of linear system solving algorithms. In: Anais da 4.a Conference of Computational Interdisciplinary Sciences. São José dos Campos, Brasil, 2016.

[14] FAHAD, A. et al. A survey of clustering algorithms for big data: Taxonomy and empirical analysis. *IEEE Transactions on Emerging Topics in Computing*, v. 2, p. 267–279, 2014.

[15] MORET, B. M.; SHAPIRO, H. D. An empirical analysis of algorithms for constructing a minimum spanning tree. *Lecture Notes in Computer Science*, v. 519, p. 400–411, 1995.

[16] GUL, M.; AMIN, N.; SULIMAN, M. An analytical comparison of different sorting algorithms in data structure. *International Journal of Advanced Research in Computer Science and Software Engineering*, v. 5, p. 1289–1298, 2015.

[17] POYREKAR, S.; NATARAJAN, S.; KERSTING, K. A deeper empirical analysis of cbp algorithm: Grounding is the bottleneck. *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*, p. 83–85, 2014.

[18] VASSALLO, M.; RALSTON, A. Algorithms for de bruijn sequences – a case study in the empirical analysis of algorithms. *The Computer Journal*, v. 35, p. 88–90, 1992.

[19] SUBRAMANI, K.; TAURAS, C.; MADDURI, K. Space-time tradeoffs in negative cycle detection - an empirical analysis of the stressing algorithm. *Applied Mathematics and Computation*, v. 215, p. 3563–3575, 2010.

[20] KHREISAT, L. Quicksort - a historical perspective and empirical study. *International Journal of Computer Science and Network Security*, v. 7, p. 54–65, 2007.

[21] DURAND, M. Asymptotic analysis of an optimized quicksort algorithm. *Information Processing Letters*, v. 85, p. 73–77, 2003.

[22] ESPELID, T. O. Analysis of a shellsort algorithm. *BIT Numerical Mathematics*, v. 13, p. 394–400, 1973.

[23] LEE, E. K.; MARTEL, C. U. When to use splay trees. *Software—Practice And Experience*, v. 37, p. 1559–1575, 2007.

[24] MALPANI, P.; BASSI, P. Analytical and empirical analysis of external sorting algorithms. In: International Conference on Data Mining and Intelligent Computing. New Delhi, India, 2014. p. 1–6.

[25] WEGBREIT, B. Mechanical program analysis. *Communications of the ACM*, v. 18, p. 528–539, 1975.

[26] MÉTAYER, D. L. ACE: an automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, v. 10, p. 248–266, 1988.

[27] FLAJOLET, P.; SALVY, B.; ZIMMERMANN, P. Lambda-Upsilon-Omega: an assistant algorithms analyzer. *Lecture Notes in Computer Science*, v. 357, p. 201–212, 1989.

[28] SILVEIRA, C. M. *Analisador de Complexidade Média Baseado nas Estruturas Algorítmicas*. Dissertação (Mestrado) — UFPEL, Pelotas, Brasil, 1998.

[29] BARBOSA, M. A. C.; TOSCANI, L. V.; RIBEIRO, L. Uma ferramenta para análise automática da complexidade de algoritmos. *Revista do CCEI*, v. 5, p. 57–65, 2001.

[30] GOLDSMITH, S. F.; AIKEN, A. S.; WILKERSON, D. S. Measuring empirical computational complexity. In: Anais da 6.a Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. Dubrovnik, Croatia, 2007. p. 395–404.

[31] BURNIM, J.; JUVEKAR, S.; SEN, K. WISE: Automated test generation for worst-case complexity. In: Anais da 31.a International Conference on Software Engineering. Vancouver, Canada, 2009. p. 463–473.

[32] NETHERCOTE, N.; SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, v. 42, p. 89–100, 2007.

[33] COPPA, E.; DEMETRESCU, C.; FINOCCHI, I. Input-sensitive profiling. *ACM SIGPLAN Notices*, v. 47, p. 89–98, 2012.

[34] ZAPARANUKS, D.; HAUSWIRTH, M. Algorithmic profiling. *ACM SIGPLAN Notices*, v. 47, p. 67–76, 2012.

[35] COSTA, E. J. et al. Um avaliador automático de eficiência de algoritmos para ambientes educacionais de ensino de programação. In: Anais da 5.a Computer on the Beach. Florianópolis, Brasil, 2014. p. 11–21.

[36] FRANCO, N. B. *Cálculo numérico*. 1.a. ed. São Paulo, Brasil: Pearson, 2006.

[37] HOFFMANN, J.; AEHLIG, K.; HOFMANN, M. Resource Aware ML. In: Anais da 24.a International Conference on Computer Aided Verification. Berkeley, USA, 2012. v. 7358, p. 781–786.

[38] HOFFMANN, J.; AEHLIG, K.; HOFMANN, M. Multivariate amortized resource analysis. *ACM SIGPLAN Notices*, v. 46, p. 357–370, 2011.

[39] MARQUARDT, D. W. An algorithm for least-squares estimation of nonlinear parameters. *Journal of The Society for Industrial and Applied Mathematics*, v. 11, p. 431–441, 1963.

[40] LEVENBERG, K. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, v. 2, p. 164–168, 1944.

[41] Gnuplot. *Gnuplot homepage*. 2018. Acesso em 17 de Abril de 2018. Disponível em: <http://www.gnuplot.info>.

[42] PETERNELLI, L. A. *Regressão Linear e Correlação*. 2003. Acesso em 20 de Março de 2017. Disponível em: <http://www.dpi.ufv.br/~peternelli/inf162.www.16032004/materiais/CAPITULO9.pdf>.

[43] YU, H.; WILAMOWSKI, B. Levenberg–Marquardt Training. In: *Industrial Electronics Handbook - Intelligent Systems*. 2.a. ed. New York, USA: CRC Press, 2011. v. 5, p. 12–1 – 12–15.

[44] GRAHAM, R. L.; KNUTH, D. E.; PATASHNIK, O. *Concrete Mathematics: a Foundation for Computer Science*. 2.a. ed. Boston, USA: Addison-Wesley Professional, 1994.

[45] HOFFMANN, J. et al. *Resource Aware ML*. 2017. Acesso em 03 de Março de 2018. Disponível em: <http://raml.co>.

[46] AZEREDO, P. *Métodos de Classificação de Dados e Análise de suas Complexidades*. Rio de Janeiro, Brasil: Campus, 1996.

[47] STRASSEN, V. Gaussian elimination is not optimal. *Numerische Mathematik*, v. 13, p. 354–356, 1969.

[48] WILLIAMS, V. V. Multiplying matrices faster than Coppersmith-Winograd. *Anais da 44.a Annual ACM Symposium on Theory of Computing*, p. 887–898, 2012.

[49] SOUZA, J. M. *Gráficos do teste do Big-Enough - WebPage*. 2018. Disponível em: <https://sites.google.com/view/graficosbe>.

[50] QIAN, N. On the momentum term in gradient descent learning algorithms. *Neural Networks*, v. 12, p. 145 – 151, 1999.

[51] BRAGA, L. P. V. *Compreendendo probabilidade e estatística*. 1.a. ed. Rio de Janeiro, Brasil: E-papers, 2010.

APÊNDICE A – Algoritmo de Levenberg–Marquardt

O *Algoritmo de Levenberg–Marquardt* (LMA) é uma interpolação entre o método de *Gauss-Newton* e o método do *Gradiente Descendente* [50]. É um método iterativo no qual, a partir de uma solução inicial arbitrária para o conjunto de parâmetros que se quer ajustar, encontra um novo conjunto de parâmetros em cada iteração, até que um critério de convergência seja atingido. Quando a solução corrente está longe da ótima, o algoritmo se comporta como o método do Gradiente Descendente (lento, mas com convergência garantida). Quando a solução corrente está próxima da ótima, o algoritmo se comporta como Gauss-Newton. O LMA, em muitos casos, encontra uma solução, mesmo que esta comece muito longe do mínimo final, tornando este método mais robusto do que o Gauss-Newton. Apesar disso, para funções bem comportadas e parâmetros iniciais razoáveis, o LMA tende a ser um pouco mais lento.

Seja $f(x, \boldsymbol{\beta})$ uma função na variável x e nos parâmetros $\beta_1, \beta_2, \ldots, \beta_k$ e seja $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_k)^T$. Portanto, $dim(\boldsymbol{\beta}) = k \times 1$. Dado um conjunto de pontos $D = \{(x_i,y_i) : 1 \leq i \leq N\}$, onde N é o número de pontos, o objetivo é encontrar os parâmetros $\hat{\boldsymbol{\beta}}$ de $f(x, \boldsymbol{\beta})$ de modo a minimizar a soma dos quadrados dos erros residuais, ou seja,

$$\hat{\boldsymbol{\beta}} = \underset{\beta}{\operatorname{argmin}}\ \mathrm{S}(\boldsymbol{\beta}), \text{ onde } S(\boldsymbol{\beta}) = \sum_{i=1}^{N} (y_i - f(x_i,\boldsymbol{\beta}))^2. \tag{4.4}$$

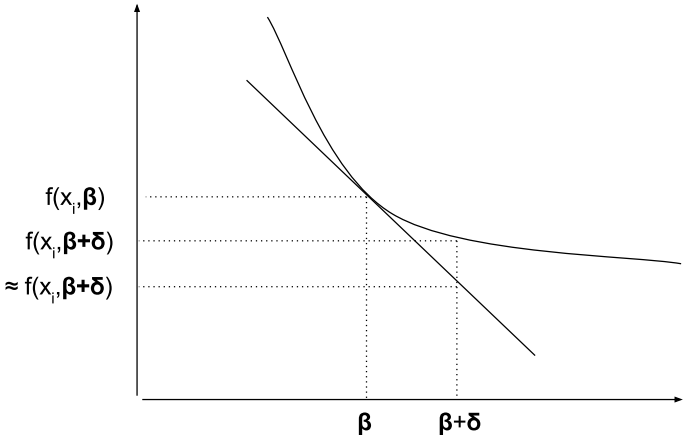
Em cada iteração, o LMA acrescenta um valor δ a β até obter um conjunto de parâmetros atingido pelo critério de convergência. Como o incremento $\boldsymbol{\delta}$ é relativamente pequeno, pode-se trocar $f(x_i,\boldsymbol{\beta} + \boldsymbol{\delta})$ pela sua aproximação linear conforme a Figura 45 e escrita como

$f(x_i,\boldsymbol{\beta} + \boldsymbol{\delta}) - f(x_i,\boldsymbol{\beta}) \approx \boldsymbol{G_i}\boldsymbol{\delta}$, onde $\boldsymbol{G_i}$ é o gradiente de $f(x_i,\boldsymbol{\beta})$, isto é,

$$\boldsymbol{G_i} = \nabla f(x_i,\boldsymbol{\beta}) = \left(\frac{\partial f(x_1,\boldsymbol{\beta})}{\partial \beta_1}(\boldsymbol{\beta}), \ldots, \frac{\partial f(x_k,\boldsymbol{\beta})}{\partial \beta_k}(\boldsymbol{\beta}) \right). \tag{4.5}$$

Portanto, $\dim(\boldsymbol{G_i}) = 1 \times k$.

Figura 45 – Aproximação linear de $f(x_i,\boldsymbol{\beta} + \boldsymbol{\delta})$.



De outra forma,

$$f(x_i,\boldsymbol{\beta}+\boldsymbol{\delta}) \approx f(x_i,\boldsymbol{\beta}) + \boldsymbol{G_i\delta}. \tag{4.6}$$

Em cada iteração, o vetor de parâmetros $\boldsymbol{\beta}$ é substituído por uma outra estimativa $\boldsymbol{\beta}+\boldsymbol{\delta}$. Substituindo (4.6) em (4.4), obtém-se

$$S(\boldsymbol{\beta}+\boldsymbol{\delta}) \approx \sum_{i=1}^{N}(y_i - f(x_i,\boldsymbol{\beta}) - \boldsymbol{G_i\delta})^2. \tag{4.7}$$

Reescrevendo em notação vetorial,

$$\begin{aligned}
S(\boldsymbol{\beta}+\boldsymbol{\delta}) \approx\ & ||\boldsymbol{y} - \boldsymbol{f(\beta)} - \boldsymbol{G\delta}||^2, \text{ onde } \boldsymbol{f(\beta)} = (f(x_1,\boldsymbol{\beta},\ldots,f(x_n,\boldsymbol{\beta}))^T \text{ e} \\
& \boldsymbol{G} = \boldsymbol{(G_1,\ldots,G_k)}^T \text{ portanto dim } (\boldsymbol{f(\beta)}) = N\times 1 \text{ e dim } (\boldsymbol{G}) = N\times k \\
=\ & \boldsymbol{(y - f(\beta) - G\delta)^T(y - f(\beta) - G\delta)} \text{ (aplicando a distributiva)} \\
=\ & \boldsymbol{(y - f(\beta))^T(y - f(\beta)) - (y - f(\beta))^TG\delta - (G\delta)^T(y - f(\beta))} \\
& \boldsymbol{+ (G\delta)^TG\delta} \text{ (se } A \text{ e } B \text{ são vetores, aplica-se a propriedade } A^TB = B^TA) \\
=\ & \boldsymbol{(y - f(\beta))^T(y - f(\beta)) - (y - f(\beta))^TG\delta - (y - f(\beta))^TG\delta} \\
& \boldsymbol{+ (G\delta)^TG\delta} \text{ (aplicando a propriedade } (AB)^T = B^TA^T) \\
=\ & \boldsymbol{(y - f(\beta))^T(y - f(\beta)) - 2(y - f(\beta))^TG\delta + \delta^TG^TG\delta},
\end{aligned}$$

onde $\boldsymbol{G}$ é a matriz *Jacobiana* cuja i-ésima linha é igual a $\boldsymbol{G_i}$ e $\boldsymbol{f}$ e $\boldsymbol{y}$ são vetores com i-ésima componente $f(x_i,\boldsymbol{\beta})$ e $\boldsymbol{y_i}$, respectivamente.

Para saber o mínimo de $S(\boldsymbol{\beta}+\boldsymbol{\delta})$, deve-se tomar a derivada em relação a variável $\boldsymbol{\delta}$ e igualar a zero (vetor nulo). Usando o fato que

$$\frac{\partial \boldsymbol{x^TAx}}{\partial \boldsymbol{x}} = \boldsymbol{x^T(A + A^T)},$$

$$\begin{aligned}
\frac{\partial S(\boldsymbol{\beta}+\boldsymbol{\delta})}{\partial \boldsymbol{\delta}} =\ & -2\boldsymbol{(y - f(\beta))^TG + \delta^T(G^TG + (G^TG)^T)} (\text{usando} (AB)^T = B^TA^T) \\
=\ & -2\boldsymbol{(y - f(\beta))^TG + \delta^T(G^TG + G^TG)} \\
=\ & -2\boldsymbol{(y - f(\beta))^TG} + 2\boldsymbol{\delta^TG^TG} = \mathbf{0}
\end{aligned}$$

Portanto,

$$\begin{aligned}
& 2\boldsymbol{\delta^TG^TG} = 2\boldsymbol{(y - f(\beta))^TG} \\
\Leftrightarrow\ & \boldsymbol{\delta^TG^T} = \boldsymbol{(y - f(\beta))^T} (\text{aplicando o transposto em ambos os lados}) \\
\Leftrightarrow\ & \boldsymbol{G\delta} = \boldsymbol{(y - f(\beta))} (\text{multiplicando por } G^T \text{em ambos os lados})
\end{aligned}$$

$$\Leftrightarrow \boldsymbol{(G^TG)\delta = G^T(y - f(\beta))}. \tag{4.8}$$

A Equação 4.8 resulta em um conjunto de equações lineares que podem ser resolvidas para encontrar $\boldsymbol{\delta}$.

Daremos um exemplo de aplicação do método. Suponha $f(x,\boldsymbol{\beta}) = \beta_1 x^{\beta_2}$, onde $\boldsymbol{\beta} = (\beta_1,\beta_2)^T$, $N = 3$, o conjunto de pontos $D = \{(0,1),(1,2),(2,5)\}$ e $\boldsymbol{\beta} = (1,1)^T$.

O erro $S(\boldsymbol{\beta})$ associado aos parâmetros $\boldsymbol{\beta}$ é dado pela Equação 4.4, logo,

$$\begin{aligned} f(x_1,\boldsymbol{\beta}) &= 1 \cdot 0^1 = 0 \\ f(x_2,\boldsymbol{\beta}) &= 1 \cdot 1^1 = 1 \\ f(x_3,\boldsymbol{\beta}) &= 1 \cdot 2^1 = 2 \\ S(\boldsymbol{\beta}) &= (y_1 - f(x_1,\boldsymbol{\beta}))^2 + (y_2 - f(x_2,\boldsymbol{\beta}))^2 + (y_3 - f(x_3,\boldsymbol{\beta}))^2 \\ &= (1-0)^2 + (2-1)^2 + (5-2)^2 = 1 + 1 + 9 = 11. \end{aligned}$$

Suponha $\boldsymbol{\delta} = (0.5,0.5)^T$. Calculemos o erro pela aproximação dada em (4.6). Os valores da função com o incremento são

$$\begin{aligned} \boldsymbol{\beta} + \boldsymbol{\delta} &= (1,1)^T + (0.5,0.5)^T = (1.5,1.5)^T \\ f(0,\boldsymbol{\beta} + \boldsymbol{\delta}) &= f(0,(1.5,1.5)) = 1.5 \cdot 0^{1.5} = 0 \\ f(1,\boldsymbol{\beta} + \boldsymbol{\delta}) &= f(1,(1.5,1.5)) = 1.5 \cdot 1^{1.5} = 1.5 \\ f(2,\boldsymbol{\beta} + \boldsymbol{\delta}) &= f(2,(1.5,1.5)) = 1.5 \cdot 2^{1.5} = 4.242. \end{aligned}$$

O erro associado aos parâmetros com incremento, aplicando-se a Equação 4.4 é dado por

$$\begin{aligned} S(\boldsymbol{\beta} + \boldsymbol{\delta}) &= (1-0)^2 + (2-1.5)^2 + (5-4.242)^2 \\ &= 1^2 + 0.5^2 + 0.758^2 \\ &= 1 + 0.25 + 0.574564 \\ &= 1.824. \end{aligned}$$

O erro aproximado, utilizando a aproximação linear, é obtido a partir da Equação 4.7.

Como $\boldsymbol{G_i} = \triangle f(x_i,\boldsymbol{\beta}) = (x_i^{\beta_2}, \beta_1 x_i^{\beta_2} \ln(x_i))$, então $\boldsymbol{G_1} = (0,0)$; $\boldsymbol{G_2} = (1,0)$; $\boldsymbol{G_3} = (2,2\ln(2))$. Portanto,

$$\begin{aligned} f(x_1,\boldsymbol{\beta}) + \boldsymbol{G_1\delta} &= 0 + (0,0)(0.5,0.5) = 0 \\ f(x_2,\boldsymbol{\beta}) + \boldsymbol{G_2\delta} &= 1 + (1,0)(0.5,0.5) = 1 + 0.5 = 1.5 \\ f(x_3,\boldsymbol{\beta}) + \boldsymbol{G_3\delta} &= 2 + (2,2\ln(2))(0.5,0.5) = 2 + 1.693 = 3.693 \\ S(\boldsymbol{\beta} + \boldsymbol{\delta}) &= (1-0)^2 + (2-1.5)^2 + (5-3.693)^2 \\ &= 1^2 + 0.5^2 + 1.307^2 = 2.958. \\ &= 2.958. \end{aligned}$$

O próximo passo é resolver o sistema linear $\boldsymbol{A\delta = B}$, onde $\boldsymbol{A} = \boldsymbol{G^T G}$ e $\boldsymbol{B} = \boldsymbol{G^T(y - f(\beta))}$ para determinar $\boldsymbol{\delta} = (\delta_1,\delta_2)$:

- Iteração #1

  $\boldsymbol{G} = ((0,0),(1,0),(2,2\ln(2)))$

  $$\boldsymbol{A} = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 2\ln(2) \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 2\ln(2) \end{pmatrix} = \begin{pmatrix} 5 & 2.772 \\ 2.772 & 1.921 \end{pmatrix}$$

  $f(\boldsymbol{\beta}) = (f(x_1,\boldsymbol{\beta}), f(x_2,\boldsymbol{\beta}), f(x_3,\boldsymbol{\beta}))^T = (0,1,2)^T$

  $\boldsymbol{y - f(\beta)} = (1,2,5)^T - (0,1,2)^T = (1,1,3)^T$

$$\boldsymbol{B} = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 2\ln(2) \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 3 \end{pmatrix} = \begin{pmatrix} 7 \\ 4.158 \end{pmatrix}$$

$$\boldsymbol{A\delta} = \boldsymbol{B} \Leftrightarrow \begin{pmatrix} 5 & 2.772 \\ 2.772 & 1.921 \end{pmatrix} \begin{pmatrix} \delta_1 \\ \delta_2 \end{pmatrix} = \begin{pmatrix} 7 \\ 4.158 \end{pmatrix}$$

$$\begin{cases} 5\delta_1 + 2.772\delta_2 = 7 \\ 2.772\delta_1 + 1.921\delta_2 = 4.158 \end{cases}$$

o que resulta em $\delta_1 = 1; \delta_2 = 0.721$.

Incrementando $\boldsymbol{\delta}$ a $\boldsymbol{\beta}$,

$\boldsymbol{\beta} = \boldsymbol{\beta} + \boldsymbol{\delta} = (1,1)^T + (1,0.721)^T = (2,1.721)^T$.

O erro associado aos parâmetros com o incremento $\boldsymbol{\delta}$, aplicando-se (4.4) é

$$\begin{aligned} f(x_1,\boldsymbol{\beta}+\boldsymbol{\delta}) &= 2 \cdot 0^{1.721} = 0 \\ f(x_2,\boldsymbol{\beta}+\boldsymbol{\delta}) &= 2 \cdot 1^{1.721} = 2 \\ f(x_3,\boldsymbol{\beta}+\boldsymbol{\delta}) &= 2 \cdot 2^{1.721} = 6.593 \\ S(\boldsymbol{\beta}+\boldsymbol{\delta}) &= (1-0)^2 + (2-2)^2 + (5-6.593)^2 \\ &= 1 + 0.25 + 2.537 \\ &= 3.787. \end{aligned}$$

- Iteração #2

  $\boldsymbol{G} = ((0,0),(1,0),(3.296,6.593\ln(2)))$

  $$\boldsymbol{A} = \begin{pmatrix} 0 & 1 & 3.296 \\ 0 & 0 & 6.593\ln(2)) \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 3.296 & 6.593\ln(2) \end{pmatrix} = \begin{pmatrix} 11.863 & 15.06 \\ 15.06 & 20.877 \end{pmatrix}$$

  $f(\boldsymbol{\beta}) = (f(x_1,\boldsymbol{\beta}), f(x_2,\boldsymbol{\beta}), f(x_3,\boldsymbol{\beta}))^T = (0,2,6.593)^T$

  $\boldsymbol{y - f(\beta)} = (1,2,5)^T - (0,2,6.593)^T = (1,0,-1.593)^T$

  $$\boldsymbol{B} = \begin{pmatrix} 0 & 1 & 3.296 \\ 0 & 0 & 6.593\ln(2) \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ -1.593 \end{pmatrix} = \begin{pmatrix} -5.250 \\ -7.279 \end{pmatrix}$$

  $$\boldsymbol{A\delta} = \boldsymbol{B} \Leftrightarrow \begin{pmatrix} 11.863 & 15.06 \\ 15.06 & 20.877 \end{pmatrix} \begin{pmatrix} \delta_1 \\ \delta_2 \end{pmatrix} = \begin{pmatrix} -5.250 \\ -7.279 \end{pmatrix}$$

  $$\begin{cases} 11.863\delta_1 + 15.06\delta_2 = -5.250 \\ 15.06\delta_1 + 20.877\delta_2 = -7.279 \end{cases}$$

  o que resulta em $\delta_1 = 0.0008; \delta_2 = -0.349$.

  Incrementando $\boldsymbol{\delta}$ a $\boldsymbol{\beta}$,

  $\boldsymbol{\beta} = \boldsymbol{\beta} + \boldsymbol{\delta} = (2,1.721)^T + (0.0008, -0.349)^T = (2.0008,1.372)^T$.

  O erro associado aos parâmetros com o incremento $\boldsymbol{\delta}$, aplicando-se a (4.4) é dado por

$$
\begin{aligned}
f(x_1,\boldsymbol{\beta}+\boldsymbol{\delta}) &= 2.0008 \cdot 0^{1.372} = 0 \\
f(x_2,\boldsymbol{\beta}+\boldsymbol{\delta}) &= 2.0008 \cdot 1^{1.372} = 2.0008 \\
f(x_3,\boldsymbol{\beta}+\boldsymbol{\delta}) &= 2.0008 \cdot 2^{1.372} = 5.178 \\
S(\boldsymbol{\beta}+\boldsymbol{\delta}) &= (1-0)^2 + (2-2.0008)^2 + (5-5.178)^2 \\
&= 1 + 0 + 0.031684 \\
&= 1.031.
\end{aligned}
$$

Levenberg [40] propôs uma alteração no método de Gauss-Newton para tornar mais rápida sua convergência, então, a Equação 4.8 foi substituída por

$$(\boldsymbol{G^T G}+\lambda \boldsymbol{I})\boldsymbol{\delta} = \boldsymbol{G^T}(\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{\beta})), \tag{4.9}$$

onde $\boldsymbol{I}$ é a matriz identidade e λ um real não-negativo chamado de *fator de amortecimento*. Por analogia, pode-se dizer que $\boldsymbol{\delta}$ é o passo em direção ao mínimo e λ controla o quão grande deve ser esse passo, onde passo significa os sucessivos incrementos dados ao vetor. A cada iteração, λ é ajustado a fim de reduzir o erro, de maneira que se λ é muito pequeno (próximo de zero), o método aproxima-se do algoritmo de Gauss-Newton. Em contrapartida, quando λ é grande, a matriz $\boldsymbol{G^T G}$ é quase diagonal, o algoritmo aproxima-se do método do Gradiente Descendente.

A desvantagem deste método proposto por Levenberg é que se o valor de λ cresce, a matriz Hessiana ($\boldsymbol{G^T G}$) calculada não é totalmente utilizada. Contudo, a Hessiana descreve a curvatura local da função f através das derivadas parciais de segunda ordem em sua diagonal, logo, é possível obter uma vantagem escalando cada componente do gradiente de acordo com a curvatura. Isso resulta em um movimento maior ao longo das direções em que o gradiente é menor para que o problema clássico do "vale do erro" não ocorra, isto é, evita que a função aproxime-se do mínimo sem nunca de fato alcançá-lo. Em função disso, alguns anos depois, Marquardt [39] propôs uma pequena alteração no método de Levenberg. A modificação consistiu em substituir a matriz identidade $\boldsymbol{I}$ pela matriz diagonal que contém os elementos diagonais de $\boldsymbol{G^T G}$, resultando no algoritmo Levenberg-Marquardt dado por

$$(\boldsymbol{G^T G} + \boldsymbol{\lambda}\ \mathrm{diag}(\boldsymbol{G^T G}))\boldsymbol{\delta} = \boldsymbol{G^T}(\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{\beta})). \tag{4.10}$$

A cada iteração o método diminui o erro em relação ao conjunto de parâmetros da iteração anterior. O método continua até que um critério de convergência seja atingido. Entre tais critérios, os principais são não alterar o λ do método em mais que certa constante de uma iteração para a próxima ou definir um número máximo de iterações.

É importante ressaltar que o Gnuplot calcula as derivadas parciais necessárias numericamente, na qual o usuário deve fornecer como entrada apenas a função, seus parâmetros e respectivos valores iniciais.

APÊNDICE B – Biblioteca do EMA

Calibração

Na etapa de calibração, o EMA utiliza a função *getSuggestedVariableValues* para sugerir uma lista de valores de variáveis, conforme os parâmetros:

- runstatement: caminho do programa $\mathcal{A}$.
- timeLowerLimit: tempo mínimo que cada execução deve gastar (em ms). Tempos muito curtos podem ser insuficientes para se determinar o perfil de execução do programa, pois tempos extras à execução do programa propriamente dito (como carregamento de máquina virtual ou de bibliotecas) podem ser significativos se o tempo de execução for muito baixo.
- timeUpperLimit: tempo máximo que cada execução deve gastar (em ms). Em geral, há uma expectativa de qual deve ser o maior valor de tempo em qualquer execução, de modo que o tempo do processo inteiro de análise não se torne inviável.
- memoryLimit: memória RAM máxima que cada execução pode consumir (em MB).
- numOfPoints: número de valores distintos de cada variável que a calibração deve produzir.
- minVarValues: valor(es) mínimo(s) para cada variável. É informado como uma lista, já que é possível ter mais que uma variável.
- maxVarValues: Valor(es) máximo(s) para cada variável. É informado como uma lista, já que é possível ter mais que uma variável.
- createInstance: nome de uma função escrita em Python que implementa a lógica de criação de uma entrada para o programa a ser analisado (programa $\mathcal{B}$).
- gapToOptimal: como foi dito, o EMA executa uma busca binária para determinar os valores das variáveis para o qual o tempo limite (timeUpperLimit) é atingido. Na prática, não é geralmente necessário progredir com a busca binária até o final e, ao invés disso, encontrar valores de variáveis para os quais o tempo de execução é próximo do tempo limite. O EMA decide encerrar a busca binária para o primeiro tempo de execução $t \leq timeUpperLimit$ tal que $\frac{timeUpperLimit-t}{timeUpperLimit} \leq gapToOptimal$. Caso não seja informado o valor deste parâmetro, é assumido o valor padrão de 0.1.

Simulação

A etapa de simulação do EMA é iniciada a partir da chamada da função *runSimulation*, de acordo com os parâmetros:

- runstatement: caminho do programa a ser analisado (programa $\mathcal{A}$).
- variableValues: Uma lista $[S_1,...,S_N]$, onde S_i é uma lista de valores discutida acima.

- createInstance: nome de um método escrito em Python que solicita a criação de uma entrada para o programa (equivale ao programa $\mathcal{B}$). Este método deve ter os seguintes parâmetros:
  - variableNames: uma lista $l = [var_1, var_2, \ldots, var_N]$ onde cada var_i são os nomes das variáveis.
  - variableValues: uma lista $l = [val_1, val_2, \ldots, val_N]$ onde cada val_i é o valor escolhido para a respectiva variável, isto é, $var_i = val_i$ para $1 \leq i \leq N$.
  - standardInput: um objeto de arquivo, onde todo conteúdo escrito neste arquivo será redirecionado para o algoritmo $\mathcal{A}$ como entrada padrão.
  - parameters: uma lista $l = [par_1, par_2, \ldots, par_p]$ dos parâmetros a serem usados para iniciar $< runstatement >$, ou seja, parâmetros de execução exigidos pelo o programa em análise.
  - usageFilename: nome do arquivo no qual o uso de recursos customizados, se houver, deve ser escrito. Neste arquivo, o uso de cada recurso deve ser relatado em uma única linha seguindo a mesma ordem de recursos personalizados que devem ser especificados em $< customResources >$ (parâmetro do método construtor que será definido na seção seguinte). Esse método deve criar uma instância de entrada para a execução de $< runstatement >$.
- samplingConvergenceFactor: consiste de par $(< recurso >, < limiar >)$. Diversas amostras são executadas sequencialmente e a média do dado recurso é atualizada depois de cada execução de nova amostra. Este processo de amostragem continua até que uma execução produza uma variação na média inferior ao limiar ou o número de amostras alcançar $< maxNumOfSamples >$.
- minNumOfSamples: número mínimo de amostras para cada valoração de variáveis.
- maxNumOfSamples: número máximo de amostras para cada valoração de variáveis.
- appending: valor lógico que indica de que forma a utilização de recursos deve ser atualizada ao arquivo. Quando *appending=False* a utilização de recursos é reiniciada (uma cópia dos valores anteriores é mantida por conveniência). Quando *appending=True*, a utilização de recurso é inserida na base atual, eventualmente sobrescrevendo uma execução com mesma valoração de variáveis se já existente na base.
- discardOutliers: uma lista de recursos para os quais *outliers* devem ser descartados para o cálculo das estatísticas. Um valor é considerado um *outlier* se estiver fora do intervalo $[mediana - 1.5 \cdot IIQ; mediana + 1.5 \cdot IIQ]$, onde IIQ significa intervalo interquartil definido como a diferença entre o quartil superior e o quartil inferior, sendo o quartil inferior a mediana dos $\lfloor (n/2) \rfloor$ menores valores de um grupo de n amostras e o quartil superior a mediana dos $\lfloor (n/2) \rfloor$ maiores valores [51].
- traceUsageOfMemoryType: tipo de memória que terá sua utilização rastreada, onde podem ser passados dois valores de parâmetro: (i) *allocated* que considera toda memória requisitada pelo programa; (ii) *loaded* que considera a memória residente, ou seja, a memória que de fato foi utilizada pelo programa. Caso o valor deste parâmetro não seja informado, o valor padrão *allocated* é considerado.

Análise

A etapa de análise do EMA é executada através dos métodos:

1. **Construtor**

   - variableNames: lista de variáveis que serão analisadas pelo EMA.
   - databaseFolder: diretório onde são gerados os arquivos que constituem a base de dados do EMA que constituem os arquivos de entrada dados ao programa sendo analisado, um arquivo com as medições de recursos e os arquivos temporários criados para executar a análise.
   - customResources: lista de recursos personalizados que são reportados pelo programa através da criação do arquivo $< usageFileName >$ (ver *createInstance* no método *runSimulation*). A quantidade medida de um recurso personalizado usado pelo programa é tomada como aquela encontrada no arquivo $< usageFileName >$ multiplicada pelo fator de escala do respectivo recurso. Portanto, um fator de escala de 1 indica que o uso de um recurso é exatamente aquele lido em $< usageFileName >$. O parâmetro *customResources* deve ser fornecido da seguinte maneira: [("NomeRecurso-1", "Unidade-1", "FatorEscala-1"), ("NomeRecurso-2", "Unidade-2", "FatorEscala-2"), ...].
   - diskInputQuota: determina a capacidade máxima (em MB) que a pasta "input" pode alcançar. Quando essa capacidade é excedida, ela é restaurada apagando-se arquivos mais antigos.

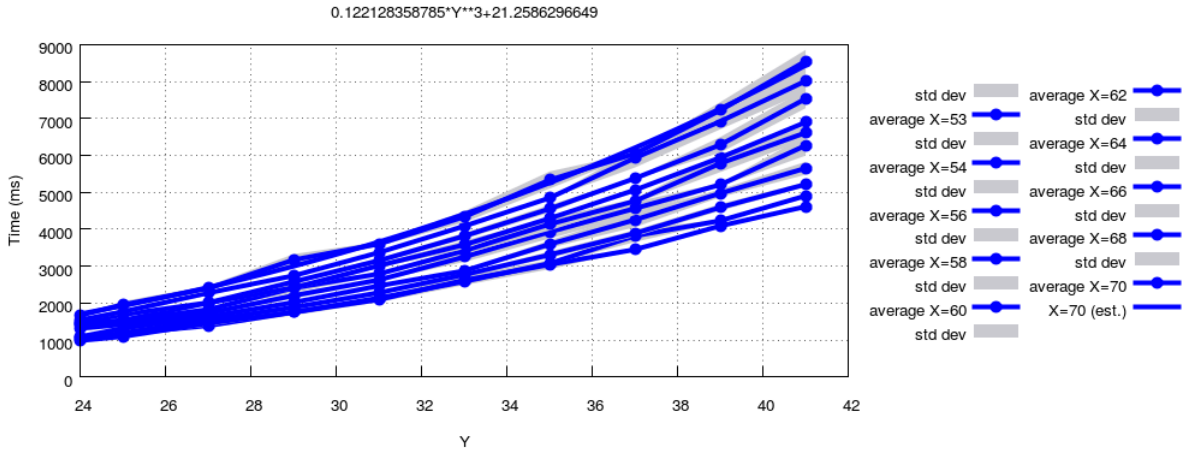
2. **estimateResourceUsage:** este método prevê o uso de recursos de determinada valoração das variáveis. Este método possui os seguintes parâmetros:

   - resource: nome do recurso a ser estimado ("Tempo", "Espaço", ou outro nome personalizado).
   - values: lista de valores de entrada para o programa a ser analisado.
   - discardTimeUnder: descarta da análise feita pelo EMA qualquer instância que levar menos tempo que aquele definido neste parâmetro (em ms).
   - equivalenceThreshold: limita o erro máximo para que cada função avaliada pelo EMA possa ser considerada equivalente àquela de mínimo erro.
   - tieBreakMaxVal: critério de desempate a ser usado para determinar as funções de utilização de recursos. Seja f a função de erro mínimo e g uma função qualquer, o $tieBreakMaxVal$ é um fator multiplicativo tal que $N* = N_{max} \cdot tieBreakMaxVal$, conforme definido na Equação 2.5, onde $N*$ é limite máximo para que uma função seja considerada viável.
   - discreteFunctionsOnly: valor lógico que indica quando devem ser consideradas apenas funções com parâmetros discretos.

3. **getResourceUsageFunction:** este método retorna uma função que estima o uso do recurso fornecido. Uma função consiste em uma lista $[t_1, \ldots, t_T]$ de tuplas onde $t_j = ([pv_1, \ldots, pv_{(N-1)}], (fnStr, p_1, p_2, \ldots, p_P, erro, [v_1, v_2, \ldots, v_P]))$ onde:

(i) pv_i é a função que corresponde ao uso do recurso quando o i-ésimo valor da variável é fixado no valor $< pv_i >$. Em outras palavras, se o uso do recurso corresponde a função F $(var_1, \ldots, var_N)$, então t_j descreve a função $G(var_N) = F(pv_1, \ldots, pv_(N-1), var_N)$. Para exemplificar, considere uma execução de um programa de duas variáveis X e Y tais que X = {53,54,56,58,60,,62,64,66, 68,70} e Y = {24,25,27,29,31,33,35,37,39,41}. Inicialmente o EMA fixa o valor de X em 53 e varia todos os pontos do conjunto Y, depois fixa o valor de X em 54 e varia todos os pontos do conjunto Y e o processo se repete para cada um dos valores de X, de maneira que a cada passo uma função melhor-palpite é retornada por este método. Os valores retornados neste parâmetro são os valores fixados. A Figura 46 mostra as curvas para cada valor de X fixado e o conjunto de pontos Y;

Figura 46 – Gráfico resultante de um programa de duas variáveis.



(ii) $fnStr$ é uma *string* de função de $G(var_N)$, onde var_N é genericamente representado por x em $< fnStr >$, ou seja, é a função melhor-palpite;

(iii) p_i são os parâmetros usados em $< fnStr >$.

(iv) $erro$ é o erro da função em $< fnStr >$;

(v) v_i é o valor dos parâmetros $< p_i >$ em $< fnStr >$.

Exemplo de retorno deste método:

[([70],(“$a_0x^3 + c_1$”,“a_0,c_1”,87579.3206033528, [0.122128358784712, 21.2586296648569]))].

Este método possui os seguintes parâmetros:

- resource: nome do recurso a ser estimado (“Tempo”, “Espaço”, ou outro nome personalizado).
- discardTimeUnder: descarta da análise feita pelo EMA qualquer instância que levar menos tempo que este parâmetro (em ms).
- case: determina o tipo de caso a ser analisado.
  - worst: maior utilização de recursos.

  - mean: utilização média de recursos.
  - best: menor utilização de recursos.
- equivalenceThreshold: limita o erro máximo para que cada função avaliada pelo EMA possa ser considerada equivalente àquela de mínimo erro (limiar de equivalência).
- tieBreakMaxVal: conforme já descrito no item 2.
- discreteFunctionsOnly: Valor lógico que indica se apenas funções discretas devem ser consideradas.
- filterFixedValues: uma lista [t_1,t_2,...], onde cada elemento também é uma lista com os valores das primeiras $N-1$ variáveis. Quando este parâmetro for informado, somente o conteúdo contendo os primeiros $N-1$ variáveis valoradas como t_i, para algum i, serão mostradas.
- printFunctionReport: Valor lógico que indica se todas as funções candidatas e suas respectivas estimativas devem ser impressas no console.

Apresentação de resultados gráficos

Os resultados das estimativas encontradas pelo EMA podem ser plotados de acordo com as especificações dos parâmetros do método *plotResourceUsage*:

- resource: nome do recurso a ser estimado ("Tempo", "Espaço", ou outro nome de recurso personalizado).
- discardTimeUnder: descarta da análise feita pelo EMA qualquer instância que levar menos tempo que este parâmetro (em ms).
- mode: altera o modo como são gerados os gráficos.
  - windows: assume uma interface gráfica.
  - term: assume um ambiente sem interface gráfica como um terminal (os gráficos são desenhados com símbolos ASCII).
- style: altera o modo como os gráficos são desenhados.
  - lines: os valores são representados como pontos conectados por segmentos de linha.
  - intervals: se mais de um caso será desenhado, eles serão conectados com uma linha vertical.
  - points: os valores são representados como pontos.
- showStdDev: valor lógico que indica se o desvio-padrão deve ser mostrado.
- cases: uma tupla de 3 binários indicando quais casos devem ser desenhados:
  - (1,0,0): pior caso.
  - (0,1,0): caso médio.
  - (0,0,1): melhor caso.

- usageFunction: função que estima a utilização de recursos. Caso seja informado, a função será desenhada no mesmo gráfico que os dados.
- exportToFolder: diretório onde serão armazenados os gráficos. Caso seja informado, os gráficos serão desenhados apenas em arquivos.
- exportToFormat: formato para a exportação dos gráficos de acordo com as opções de exportação do Gnuplot.
- appendToPlot: uma cadeia de caracteres retornada pelo *plotResourceUsage*() representando um gráfico ao qual o gráfico atual deve ser desenhado junto.
- multiplotTitle: texto que será fixado em cada camada do gráfico, para que permaneçam distintas mesmo após a sua junção.
- autoColor: ao usar o *multiplotting*, o parâmetro autoColor determina se cada novo argumento informado deve ter sua cor definida automaticamente ou, caso contrário, deve manter a especificação de cores original de cada novo argumento informado.
- customColors: quando o autoColor está desativado, este parâmetro altera as cores padrão. O valor deve ser um dicionário {"verde": <nova cor>, "azul": <nova cor>, "vermelho": <nova cor>}.
- keyPosition: onde as chaves devem ser colocadas ('X Y', onde X em {'top', <'center'>, 'abaixo'} e Y = 'esquerda' / 'centro' / <'direita'>).
- size: (altura, largura) do gráfico.
- fontFamily: (nome, tamanho) da fonte para escrever as legendas.
- lineWidth: largura da curva de estimativa.
- rangeTics: define uma faixa ((xmin,xmax),(ymin,ymax)) para plotagem do gráfico. Para plotar todos os pontos, basta informar *"None"* .
- scientificDecimals: (<número de decimais para x >, <número de decimais para y >); usa-se *None* para um determinado número de decimais para deixar um determinado eixo no modo automático.
- translations: dicionário de pares de *strings* <original,tradução> para legendas automáticas (por exemplo: tempo, espaço, média, entre outros.)
- labelsEncoding: codificação utilizada em *strings* para legendas.

APÊNDICE C – Implementações de Algoritmos

BubbleSort em C

```
1
2
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  void bubble_sort(int size, int *data) {
7    int *a;
8    int *b;
9
10   for(a=data; a<data+size; ++a) {
11     for(b=a+1; b<data+size; ++b) {
12       if (*a > *b) {
13   *a ^= *b;
14   *b ^= *a;
15   *a ^= *b;
16       }
17     }
18   }
19 }
20
21
22 int main() {
23   int * L;
24   int N, i;
25
26   scanf("%d", &N);
27   L = (int *)(malloc(N*sizeof(int)));
28
29   for (i = 0; i < N; i++) {
30     L[i] = N-i;
31   }
32
33   bubble_sort(N, L);
34   free(L);
35
36   return 0;
37 }
```

MergeSort em C

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include <time.h>
4
5  void Merge(int L[], int s, int m, int e) {
6    int * Lc;
7    Lc = (int *)(malloc((e-s+1)*sizeof(int)));
8    int i, j, k;
9    i = s; j = m+1;
10   for (k = 0; k < e-s+1; k++) {
11     if ((j > e) || (i <= m && L[i] <= L[j])) {
12       Lc[k] = L[i];
13       i = i+1;
14     } else {
15       Lc[k] = L[j];
16       j = j+1;
17     }
18   }
```

```
19    for (k = 0; k < e-s+1; k++) {
20      L[s+k] = Lc[k];
21    }
22    free(Lc);
23  }
24
25  void MergeSort(int L[], int s, int e) {
26    if (s < e) {
27      int m;
28      m = (s+e)/2;
29      MergeSort(L,s,m);
30      MergeSort(L,m+1,e);
31      Merge(L,s,m,e);
32    }
33  }
34
35  int main() {
36    int * L;
37    int N, i;
38
39    srand(time(NULL));
40    scanf("%d", &N);
41    L = (int *)(malloc(N*sizeof(int)));
42
43    for (i = 0; i < N; i++) {
44      L[i] = N-i;
45    }
46    for (i = 0; i < N; i++) {
47      int j, t;
48      j = rand() \% N;
49      t = L[i]; L[i] = L[j]; L[j] = t;
50    }
51
52    MergeSort(L, 0, N-1);
53    free(L);
54
55    return 0;
56  }
```

Quicksort em OCaml

```
1  let rec append l1 l2 =
2    match l1 with
3      | [] -> l2
4      | x::xs -> x::(append xs l2)
5
6
7  let rec partition p l =
8    match l with
9      | [] -> ([],[])
10     | x::xs ->
11       let (cs,bs) = partition p xs in
12         if p < (x:int) then
13           (cs,x::bs)
14         else
15           (x::cs,bs)
16
17
18 let rec quicksort l =
19   match l with
20   | [] -> []
21   | x::xs ->
22       let (ys, zs) = partition x xs in
23         append (quicksort ys) (x :: (quicksort zs));;
```

Quicksort em OCaml com recursão de cauda

```
let rec trev l r =
  match l with
    | [] -> r
    | x::xs -> trev xs (x::r);;
let rev l = trev l [];;

let rec tunir l1 l2 r =
  match l1 with
    | [] -> (match l2 with
        | [] -> rev r
        | xl2::xsl2 -> tunir [] xsl2 (xl2::r))
    | x1::xs1 -> tunir xs1 l2 (x1::r);;
let unir l1 l2 = tunir l1 l2 [];;

let rec tpart x l l1 l2 =
  match l with
    | [] -> (match l1 with
          | [] -> ((x::[]), l2)
          | xl1::xsl1 ->(l1, (x::l2)) )
    | (lx:: lxs) -> if lx <= (x:int) then
              tpart x lxs (lx::l1) l2
            else
              tpart x lxs l1 (lx::l2);;
let part x l = tpart x l [] [];;

let rec tnroelem l n =
  match l with
    | [] -> n
    | x::xs -> tnroelem xs (n+1);;
let nroelem l = tnroelem l 0;;

let rec tunirL l r =
  match l with
    | [] -> rev r
    | lx::lxs -> (match lx with
          | [] -> tunirL lxs r
          | xlx::xslx -> tunirL(xslx::lxs) (xlx::r));
let unirL l = tunirL l [];;

let rec tquicksort lm l lM =
  match l with
    | [] -> unirL (unir (rev lm) lM)
    | lx::lxs -> (match lxs with
            | [] -> unirL (unir (rev (l :: lm)) lM)
            | xlxs::xslxs -> let (la, lb) = part lx lxs in
                    if (nroelem la < nroelem lb) then
                        tquicksort ((quicksort la)::lm) lb lM
                    else
                        tquicksort lm la ((quicksort lb)::lM) )
and quicksort l = tquicksort [] l [];;
```

A seguir, encontra-se uma descrição breve de cada função. A função "t<função>" corresponde à implementação da função "<função>" com recursão de cauda.

- rev: recebe como parâmetro uma o uma lista a ser invertida.
- unir: recebe como parâmetro duas listas $l1$ e $l2$ e retorna uma lista que contém os elementos de $l1$ e $l2$.
- part: recebe como parâmetro um elemento pivô e uma lista que a ser particionada em duas sublistas, uma com os elemento menores que o pivô e a outra com os elementos maiores.
- nroelem: recebe uma lista na qual deseja-se obter o número de elementos da lista passada como parâmetro.

- unirL: recebe como parâmetro uma lista l de listas l_i na qual deseja-se obter uma lista com todos os elementos de cada lista l_i.
- quicksort: recebe uma lista na qual deseja-se ordenar (função principal).

Busca em profundidade em OCaml

```
(* Rnat *)
type t = int

let zero : t = 0
let succ (i : t) : t = i + 1

let to_int (i : t) : int = i
let of_int (i : int) : t =
  if i < 0 then
    raise (Invalid_argument "Rnat.of_int : negtive integer")
  else
    i

let add (a : t) (b : t) : t = a + b
let mult (a : t) (b : t) : t = a * b
let minus (a : t) (b : t) : t * t =
  if a <= b then
    raise (Invalid_argument "Rnat.minus : underflow")
  else
    (a - b, b)
let minusc (c : int) (n : t) : t * t =
  if n <= c then
    raise (Invalid_argument "Rnat.minusc : underflow")
  else
    (n - c, c)
let div_mod (a : t) (b : t) = (a / b, a mod b, b)

let ifz (n : t) (then_ : unit -> 'a)  (else_ : t -> 'a) : 'a =
  if n = 0 then then_ () else else_ (n - 1)

(* Rarray*)

type 'a t = 'a array

let make = Array.make
let create = Array.make

let set = Array.set
let get = Array.get
let length = Array.length

(* Programa *)

let criarGrafo n =
  (make (succ n) [], make (succ n) []);;

let criarListaAdjv g v ladjv =
  let (la, m) = g in
  set la v ladjv;;

let marcar g v =
  let (la, m) = g in
  set m v [1];;

let marcado g v =
  let (la, m) = g in
  let l = get m v in
  match l with
  | [] -> false
  | x::xs -> true;;

let viz g v =
  let (la, m) = g in
```

```
64    get la v;;
65
66  let rec buscap g v l ladjv =
67    let _ = marcar g v in
68    match ladjv with
69    | [] -> l
70    | w::ws -> let llin = (v, w)::l in
71      if marcado g w then
72        buscap g v llin ws
73      else
74        let llin2 = buscap g w llin (viz g w) in
75              buscap g v llin2 ws;;
76
77  let rec bpc g v l =
78    ifz v
79    (fun () -> l)
80    (fun v ->
81      if marcado g v then
82        bpc g v l
83      else
84        let llin = buscap g v l (viz g v) in
85        bpc g v llin
86
87    );;
```