ANÁLISE ESTÁTICA DE ALGORITMOS COM GERAÇÃO DE COMPLEXIDADE ASSINTÓTICA

STATIC ANALYSIS OF ALGORITHMS WITH ASYMPTOTIC COMPLEXITY GENERATION

Felipe Destro Munhos SILVA
felipedestro8@gmail.com
Ciência da Computação, Unianchieta

Giovanni Fernandes VICENTIN
giovannifvicentin@gmail.com
Ciência da Computação, Unianchieta

Clayton Augusto VALDO
clayton.valdo@anchieta.br
Ciência da Computação, Unianchieta

Resumo
Este estudo teve como meta validar e evidenciar, na prática, a aplicação da complexidade assintótica por meio da análise estática de algoritmos. Para isso, foi desenvolvida uma ferramenta web interativa que possibilita a geração e visualização da complexidade de diversos algoritmos, tornando esses conceitos teóricos, frequentemente considerados abstratos, mais acessíveis, concretos e passíveis de verificação em um contexto prático. A pesquisa abordou desde os fundamentos da teoria da complexidade algorítmica, passando pela aplicação de heurísticas de identificação de estruturas de controle, até a construção de representações estruturais como Árvores Sintáticas Abstratas (AST) e Grafos de Fluxo de Controle (CFG), que serviram de base para a inferência da complexidade. A partir dessa fundamentação, foi desenvolvido o sistema Big O Analyzer, utilizando tecnologias modernas como Next.js e Monaco Editor, o que permitiu integrar uma interface interativa a um pipeline de análise eficiente e modular. O sistema processa trechos de código escritos em linguagens como JavaScript, Python e Java, identifica padrões como loops, recursões e divisões de problema e retorna a classe de complexidade correspondente, acompanhada de explicações textuais e sugestões de otimização. Os resultados mostraram que a aplicação foi capaz de reconhecer corretamente diferentes classes de complexidade de O(1) a O(n!) demonstrando consistência com os fundamentos teóricos e confirmando a validade da proposta. Além de sua utilidade prática, o projeto apresenta relevância acadêmica ao oferecer uma nova abordagem de ensino para o estudo de algoritmos, unindo interatividade, visualização e formalismo teórico.

Palavras-Chave
análise estática; complexidade assintótica; notação Big O; AST; CFG; Next.js; algoritmos.

Abstract
This study aimed to validate and demonstrate, in practice, the application of asymptotic complexity through the static analysis of algorithms. To achieve this, an interactive web tool was developed that enables the generation and visualization of the complexity of various algorithms, making these theoretical concepts often considered abstract more accessible, concrete, and verifiable in a practical context. The research covered topics ranging from the fundamentals of algorithmic complexity theory, the application of heuristics for identifying control structures, to the construction of structural representations such as Abstract Syntax Trees (AST) and Control Flow Graphs (CFG), which served as the foundation for complexity inference. Based on this groundwork, the Big O Analyzer system was developed using modern technologies such as Next.js and Monaco Editor, allowing the

integration of an interactive interface with an efficient and modular analysis pipeline. The system processes code snippets written in languages such as JavaScript, Python, and Java, identifies patterns such as loops, recursion, and problem division, and returns the corresponding complexity class, accompanied by textual explanations and optimization suggestions. The results showed that the application was able to correctly recognize different complexity classes, from O(1) to O(n!), demonstrating consistency with theoretical foundations and confirming the validity of the proposal. In addition to its practical usefulness, the project holds academic relevance by offering a new teaching approach for the study of algorithms combining interactivity, visualization, and theoretical formalism.

Keywords
static analysis; asymptotic complexity; Big O notation; AST; CFG; Next.js; algorithms.

INTRODUÇÃO

Entre as décadas de 1960 e 1970, a discussão sobre eficiência de algoritmos ganhou estatuto formal com a publicação de The Art of Computer Programming, de Donald Knuth, que sistematizou fundamentos e ajudou a popularizar a notação assintótica (Big O) como linguagem padrão para comparar ordens de crescimento, independentemente de plataforma ou linguagem. Ao mesmo tempo, Knuth (1997), enfatizou a utilidade de raciocínios aproximados para fins de comparação: “We often want to know a quantity approximately, instead of exactly, in order to compare it to another.”. Esse princípio reforça que a notação assintótica privilegia ordens de crescimento sobre constantes e detalhes de implementação. Desde então, a notação assintótica firmou-se como padrão de fato para medir e comunicar o desempenho de algoritmos na computação moderna.

Na prática, consolidou-se o uso de microbenchmarks e profiling pós-implementação para comparar algoritmos; são úteis, mas sujeitos a vieses de hardware, compilador/JIT e distribuição das entradas, o que frequentemente gera leituras parciais. Como lembra Dijkstra (1970), “Program testing can be used to show the presence of bugs, but never to show their absence!”, uma advertência que, por analogia, vale para desempenho: testes revelam gargalos, não provam a classe de complexidade.

Por isso, ganhou força a aproximação da análise assintótica ao próprio código via análise estática, produzindo estimativas fundamentadas e reprodutíveis que complementam, em vez de substituir, a experimentação. Nesse cenário, a análise estática consolidou técnicas e ferramentas que extraem propriedades sem executar o programa de AST/CFG a interpretação abstrata e análise de fluxo de dados formando uma base sólida para inferência automática de custos assintóticos e explicações reproduzíveis. Assim, aproximar a complexidade do próprio código deixa de ser apenas uma conveniência didática e torna-se um caminho concreto para padronizar comparações e reduzir vieses de medição. Conforme sintetiza a JETBRAINS (2025), a análise estática examina o código sem execução, detecta precocemente defeitos e vulnerabilidades e, quando integrada ao CI/CD, atua de forma complementar à análise dinâmica.

A proposta deste artigo é investigar técnicas de análise estática de código para estimar a complexidade assintótica de algoritmos. A proposta envolve a construção de um sistema que, a partir de trechos de código, gera árvores sintáticas abstratas (AST) e grafos de fluxo de controle (CFG), aplicando regras heurísticas para identificar estruturas críticas como recursão, loops aninhados e algoritmos de ordenação, fornecendo justificativas detalhadas da classificação obtida. Do ponto de vista prático, um sistema desse tipo tem duas aplicações centrais: ensino de algoritmos, ao tornar visível a ligação entre estruturas de controle e ordens de crescimento com justificativas locais e gráficos; e ecossistema de compiladores e revisão de código, ao fornecer alertas precoces sobre potencial degradação assintótica e pistas de otimização ainda na fase de desenvolvimento, sem dependência de cargas de teste específicas.

FUNDAMENTAÇÃO TEÓRICA

Com base nos conceitos apresentados, se originou a ideia de criar um sistema web para análise

estática de algoritmos. A ideia é utilizar ferramentas modernas e confiáveis, que auxiliem tanto na aprendizagem de algoritmos quanto na prática, servindo como um recurso valioso para compiladores. Para tanto, foi realizada uma pesquisa minuciosa a fim de se compreender melhor a fundamentação teórica e o funcionamento de como se faz a análise de algoritmos em função do crescimento assintótico.

Teoria da Complexidade Algorítmica

Podemos nos deparar em algumas ocasiões com problemas e/ou dificuldades de aplicabilidade quando trabalhamos com algoritmos, isso é relacionado ao seu tamanho de entrada ou complexidade, é de extrema importância compreender a complexidade do algoritmo com o qual estamos lidando. Segundo (Cormen et al, 2009): "Algoritmos diferentes criados para resolver o mesmo problema muitas vezes são muito diferentes em termos de eficiência. Essas diferenças podem ser muito mais significativas que as diferenças relativas a hardware e software.". Essa ideia demonstra como a análise de complexidade é essencial, pois ajuda a deixar de lado detalhes específicos de hardware e implementação. Com base nisso, podemos comparar diferentes soluções para o mesmo problema de forma clara.

Nessas circunstâncias, a notação assintótica consolidou-se como princípio de referência para descrever o comportamento de algoritmos, do tempo de execução e do consumo de memória à medida que o tamanho da entrada cresce. Como citado por Sipser (2013), "The theories of computability and complexity are closely related. In complexity theory, the objective is to classify problems as easy ones and hard ones; Whereas in computability theory, the classification of problems is by those that are solvable and those that are not. Computability theory introduces several of the concepts used in complexity theory", ela também destaca que classificar problemas e algoritmos de acordo com diferentes níveis de crescimento é importante para entender o quão difícil eles são por natureza.

Além disso, é importante considerar que a análise de complexidade se divide em dois principais eixos: tempo e espaço. A complexidade de tempo mede a quantidade de operações necessárias para concluir a execução de um algoritmo, enquanto a complexidade de espaço mede a quantidade de memória utilizada durante o processo. Muitas vezes, existe um equilíbrio entre tempo e espaço, conhecido como trade-off, em que o aumento da eficiência temporal pode implicar maior consumo de memória, e vice-versa. Essa relação é descrita por (Knuth, 1997) como um dos aspectos centrais da eficiência algorítmica, pois qualquer otimização deve considerar ambos os fatores para alcançar o melhor desempenho possível.

(Cormen et al., 2009) ressaltam que as principais classes de complexidade podem ser organizadas em categorias específicas, conforme mostrado na Tabela 1. Essa tabela apresenta as ordens de crescimento mais comuns, bem como exemplos significativos de algoritmos relacionados. Essa classificação ajuda a comparar desempenho e prever escalabilidade à medida que o tamanho da entrada cresce.

Tabela 1. Classes de Complexidade Assintótica (Bigocheatsheet, 2013)

Classe	Crescimento	Exemplo clássico
O(1)	Constante	Acesso direto em vetor
O(log n)	Logarítmico	Busca binária
O(n)	Linear	Percorrer lista
O(n log n)	Linearítmico	Merge Sort
$O(n^2)$	Quadrático	Bubble Sort
$O(2^n)$	Exponencial	Backtracking
O(n!)	Fatorial	Permutações completas

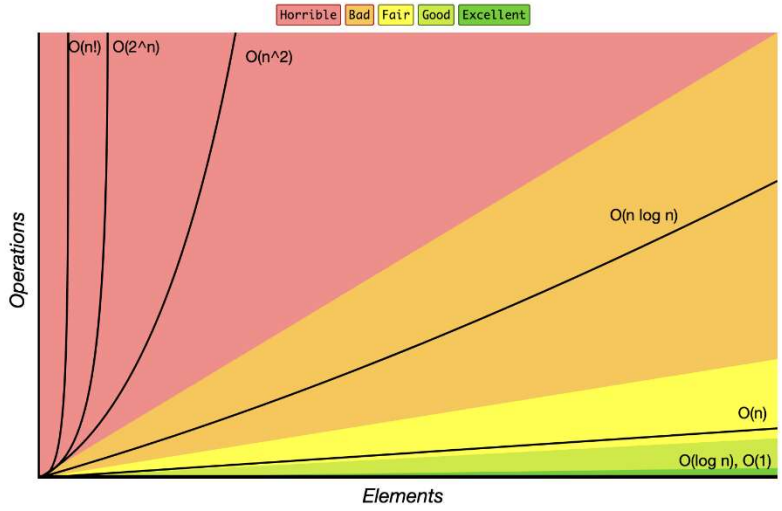
Além disso, a teoria da complexidade engloba a categorização dos problemas computacionais em classes amplas e teoricamente fundamentadas, como P, NP e NP-completo, de acordo com Sipser (2013). Os problemas que pertencem à classe P podem ser solucionados em tempo polinomial. Em

contrapartida, os problemas da classe NP permitem soluções que podem ser verificadas em tempo polinomial, mas ainda não se sabe se todos esses problemas podem ser resolvidos diretamente nesse mesmo intervalo de tempo. Essa distinção é fundamental para compreender o que é eficiente ou intratável na computação atual, sendo um dos alicerces da análise teórica de algoritmos.

Assim, a análise assintótica consolida-se permitindo decisões mais racionais sobre qual abordagem adotar em diferentes contextos. Como afirmam (Cormen et al., 2009): "Analisar um algoritmo significa prever os recursos de que o algoritmo necessita. Ocasionalmente, recursos como memória, largura de banda de comunicação ou hardware de computador são a principal preocupação, porém mais frequentemente é o tempo de computação que desejamos medir. Em geral, pela análise de vários algoritmos candidatos para um problema, pode-se identificar facilmente um que seja o mais eficiente. Essa análise pode indicar mais de um candidato viável, porém, em geral, podemos descartar vários algoritmos de qualidade inferior no processo." Essa abordagem mostra que a teoria da complexidade algorítmica não é só uma ideia vaga, mas também um guia útil para distinguir entre soluções mais eficazes e aquelas que são menos eficientes.

Tal estrutura pode ser observada na Figura 1, que ilustra graficamente as diferentes ordens de crescimento em função do tamanho da entrada.

Figura 1. Gráfico de Complexidade Big-O (Bigocheatsheet, 2013)



Heurísticas de Classificação de Estruturas de Controle

A análise da complexidade assintótica pode ser realizada por meio de heurísticas que conectam estruturas de controle no código a padrões de crescimento computacional. Essas heurísticas funcionam como normas aplicadas que permitem estimar a ordem de complexidade apenas pela sintaxe do programa, eliminando a necessidade de fazer cálculos matemáticos tradicionais em muitos casos. Conforme a figura 2, essa estratégia é essencial não apenas no ensino de algoritmos, pois torna-se mais fácil academicamente de se entender, mas também nas ferramentas de análise estática de código, que empregam árvores sintáticas e grafos de fluxo de controle para detectar laços, recursões e ramificações que afetam o desempenho (Aho et al., 2008).

Em um modo geral, loops simples representam o primeiro nível de classificação. Estruturas de repetição, como for ou while, ao percorrerem a entrada de uma forma linear exibem uma complexidade igual O(n), pois cada iteração processa um pedaço que é proporcional ao tamanho da entrada (Cormen et al., 2009). A heurística fundamental é: um laço→ $O(n)$, indicando um crescimento linear.

No próximo nível, loops aninhados indicam a composição multiplicativa do espaço de iterações. Por exemplo, dois laços for aninhados até n levam a $O(n^2)$, já que cada elemento da entrada é tratado em combinação com todos os demais. Essa conexão pode ser estruturada em tabelas hierárquicas, nas quais cada nível de aninhamento representa uma nova potência de n (Sedgewick e

Wayne, 2011).

Um padrão importante é também a estrutura recursiva. A recursão simples, como no exemplo do fatorial, resulta em $O(n)$. Já a recursão do tipo Divide and Conquer, como no MergeSort ou QuickSort, segue a relação de recorrência $\mathrm{T(n)} = 2\mathrm{T}(n/2) + \mathrm{O(n)}$, cuja solução resulta em $O(n \log n)$ (Cormen et al., 2009), onde tal resultado é apresentado na figura 3. Isso serve para ilustrar a importância das árvores de recursão, que podem ser representadas graficamente para mostrar como o problema é dividido de forma sucessiva.

Além das estruturas iterativas e recursivas, heurísticas também podem ser aplicadas a estruturas condicionais e blocos dependentes, que interferem o custo total de execução conforme o número de caminhos possíveis. Estruturas condicionais, embora não representem repetição explícita, assim como impactam a complexidade ao definir caminhos alternativos de execução. A heurística clássica assume o pior caso (o caminho mais custoso) como representativo da ordem de crescimento.

Em CFGs, esse comportamento é refletido por múltiplos caminhos de controle, da qual união determina o limite superior assintótico. Assim, a análise estática tende a aproximar o custo total pela soma ou máximo dos fluxos de controle, conforme o modelo de execução adotado. Esse tipo de inferência é singularmente relevante em códigos com múltiplas dependências condicionais ou com funções que possuem comportamento variável em tempo de execução (LLVM PROJECT, 2018).

A interpretação das heurísticas também muda dependendo do paradigma de programação. Em linguagens imperativas, o controle de fluxo explícito possibilita estimativas diretas tanto de iteração quanto de recursão. No caso de linguagens orientadas a objetos, a análise deve levar em conta o custo das chamadas dinâmicas e o encadeamento de métodos que são herdados. Em vez de loops, encontramos recursões puras e composições de funções em linguagens funcionais, o que exige heurísticas para reconhecer padrões como map, reduce ou fold no lugar de loops explícitos. Essas variações ressaltam a necessidade de alinhar o modelo heurístico com o paradigma em questão, a fim de obter maior precisão e compatibilidade entre diversos estilos de programação (Kosinski et al., 2024).

Por fim, heurísticas mais detalhadas se concentram em padrões clássicos, como a busca binária $O(\log n)$, loops duplos dependentes $O(n * m)$ ou algoritmos de tempo exponencial $O(2^n)$, que normalmente envolvem uma combinação exaustiva. A ordenação desses modelos é um passo importante tanto no ensino de algoritmos quanto na prática profissional, pois ajuda a achar rápido o crescimento pela estrutura do código. Desse modo, Sipser (2013) ressalta que a análise do tempo de execução e o reconhecimento de problemas intratáveis são fundamentais para compreender os limites da computação e situar algoritmos dentro de diferentes níveis de complexidade.

Figura 2. Operações Comuns de Estruturas de Dados (Bigocheatsheet, 2013)

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Figura 3. Algoritmos de Ordenação de Arrays (Bigocheatsheet, 2013)

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

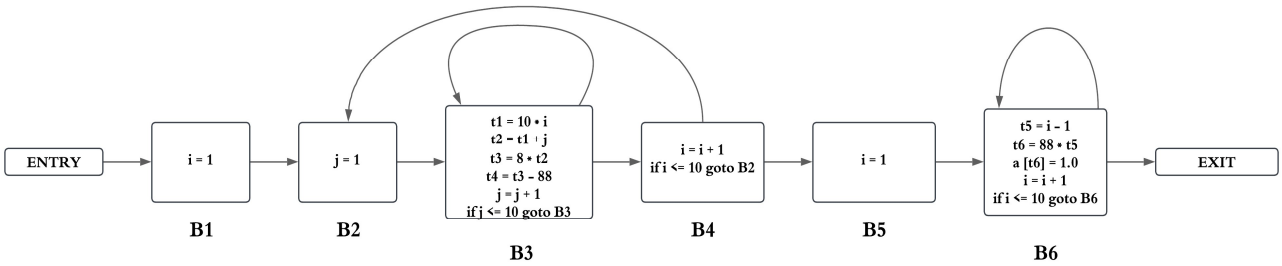
Análise Estática de Código (AST/CFG)

A análise estática investiga propriedades do programa sem executá-lo, traduzindo o código-fonte em representações que tornam explícitas a estrutura e o fluxo lógico. Duas peças centrais são a Árvore Sintática Abstrata (AST), que preserva o conteúdo semanticamente relevante do programa, e o Grafo de Fluxo de Controle (CFG), que mapeia os caminhos possíveis de execução entre blocos básicos, de que modo é exibido na figura 4. Em termos simples, a AST descreve o que o programa é, enquanto o CFG mostra como ele pode ser percorrido. Sobre CFG, a documentação do GCC define: "The CFG is a directed graph where the vertices represent basic blocks and edges represent possible transfer of control flow from one basic block to another." Esse par de estruturas, consolidado na literatura de compiladores e análise de programas, sustenta estimativas assintóticas por evidenciar padrões estruturais que influenciam diretamente o custo (GCC., 2025).

A partir dessas representações, a ferramenta identifica hierarquias de aninhamento, recursões e composições frequentes na AST, enquanto o CFG explicita ciclos e caminhos mutuamente exclusivos em condicionais. Essas informações são essenciais para raciocinar sobre limites de iteração e composição de custos. Para refinar as inferências sem executar o programa, aplicam-se análises de fluxo de dados e a interpretação abstrata, que propagam invariantes, como avançar um índice até n ou uma recursão que divide o problema por 2, e permitem construir limites superiores de forma sistemática. Na camada estrutural, a forma Static Single Assignment (SSA) e o grafo de dependência de controle simplificam o encadeamento de definições e usos e a relação de dominação entre blocos, vale lembrar que "LLVM is a Static Single Assignment (SSA) based representation" (LLVM PROJECT, 2018).

No sistema proposto, o pipeline conceitual segue parsing, construção da AST, derivação do CFG, análises de dados e controle, aplicação de regras de complexidade e emissão de justificativas. A saída é explicável e reprodutível: a classe Big-O, os pontos críticos que a sustentam e as hipóteses adotadas sobre o que é n e sobre pior caso. Em implementações modernas de IR em SSA, limitações práticas, como custos não visíveis de bibliotecas, polimorfismo e reflexão, ou limites dependentes de dados de entrada podem ser tratadas com resumos de custo conhecidos e, quando necessário, com intervalos ou anotações do usuário. Na literatura recente, "Static program analysis, or static analysis, aims to discover semantic properties of programs without running them." (RIVAL; YI, 2020). Além disso, há propostas contemporâneas que certificam limites superiores de custo para reforçar a reprodutibilidade dos resultados (ALBERT et al., 2025).

Figura 4. Exemplo de Grafo de Fluxo de Controle (Aho et al., 2008)



A análise estática de código é um campo consolidado na engenharia de software e na teoria da computação, fundamentado em métodos formais que examinam o comportamento potencial de um programa a partir de sua estrutura sintática. Diferentemente da análise dinâmica, que depende da execução e das condições de entrada, a análise estática utiliza modelos abstratos para deduzir propriedades determinísticas e reprodutíveis, como alcance de variáveis, dependências de controle e limites de iteração (Nielson et al., 1998).

Entre as técnicas mais recorrentes estão a interpretação abstrata e a análise de fluxo de dados, que permitem inferir estados possíveis do programa por meio da propagação de informações entre blocos básicos. A interpretação abstrata, proposta por Cousot (1977), oferece uma estrutura matemática para representar o comportamento do código em domínios simplificados como intervalos, sinais ou contadores de iteração garantindo que as conclusões obtidas sejam seguras, ainda que aproximadas. Isso a torna ideal para inferir limites assintóticos de laços e recursões, já que o objetivo não é calcular um resultado exato, mas determinar o crescimento relativo das operações em função do tamanho da entrada.

O uso combinado de AST e CFG constitui a base prática dessas análises. A AST fornece a estrutura. No ciclo de vida moderno, a análise estática é frequentemente automatizada em CI/CD, executando em cada commit para manter qualidade, segurança e desempenho com feedback rápido à equipe, ferramentas como o Qodana (JetBrains) documentam essa integração e seus benefícios (inspeções consistentes, configuração de severidades, relatórios centralizados), reforçando o papel da análise estática (JETBRAINS, 2025).

Plataformas Web Interativas

Desde o fim dos anos 1980, quando Tim Berners-Lee propôs no CERN um sistema de hipertexto distribuído para organizar e compartilhar informações científicas, “It discusses the problems of loss of information about complex evolving systems and derives a solution based on a distributed hypertext system.”. A Web passou de páginas estáticas acessadas por navegadores simples a um ecossistema padronizado e amplamente interoperável, como demonstra na figura 5, porém, só em 2014 o HTML5 se consolidou como recomendação do W3C, unificando funcionalidades essenciais como multimídia, gráficos e APIs, estabelecendo a base para aplicações ricas no navegador. Nesse contexto histórico-técnico, torna-se natural evoluir para plataformas web interativas que integram front-end e back-end com renderização híbrida e ferramentas de edição em tempo real. (BERNERS-LEE, 1989; W3C, 2014).

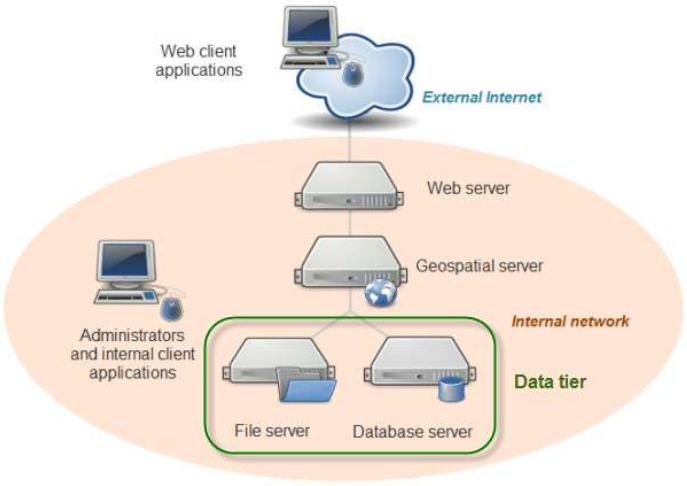
Ao longo dos anos, o desenvolvimento web tem sofrido diversas mudanças e otimizações, e nesse processo surgiram frameworks e bibliotecas mais performáticas, com foco em desempenho e manutenibilidade. Inserido nesse contexto, a proposta materializa-se em uma plataforma web construída com Next.js, framework full-stack sobre Node, React, integra frontend e backend na mesma aplicação, com roteamento por arquivos, rotas de API e renderização híbrida. O editor embutido utiliza o Monaco Editor, que, segundo a documentação oficial, é "the code editor that powers VS Code". Esse arranjo permite ao usuário escrever e editar um código, enviar para análise e visualizar imediatamente a classe de complexidade com gráficos e explicações, sem sair da página (Microsoft/Monaco, 2025).

O fluxo da aplicação é objetivo, a página envia o código e metadados, por exemplo, qual coleção define n ao serviço de análise por um Route Handler, “Route Handlers allow you to create custom request handlers for a given route using the Web Request and Response APIs.” O backend constrói a AST e o CFG, aplica as regras e retorna a classe Big-O com uma prova textual e artefatos para visualização. Para não bloquear a interface, o front usa streaming com Server Components, “Streaming: Server Components allow you to split the rendering work into chunks and stream them to the client as they become ready.”. A UI exibe gráficos de crescimento e destaca, em linguagem simples, porque determinado laço ou recursão implica a ordem reportada. Por segurança e reprodutibilidade, o sistema não executa o código submetido; toda a inferência decorre de parsing e análise estática fornecida pelo usuário (Next.js, 2025).

Do ponto de vista técnico, o front-end atua como uma camada de interação, responsável por capturar o código digitado pelo usuário e renderizar visualmente os resultados, como árvores, grafos e explicações textuais. Já o back-end executa o núcleo da análise estática, aplicando algoritmos de parsing, derivação sintática e inferência de complexidade. O resultado é uma aplicação modular, na qual a camada de exibição é independente da lógica analítica, garantindo portabilidade e reuso em diferentes contextos de ensino e pesquisa (Adamkó, 2014).

Outro aspecto relevante é a reprodutibilidade dos resultados. Por não executar o código analisado, mas apenas inspecionar sua estrutura sintática e lógica, o sistema garante que o comportamento observado seja independente do ambiente e dos dados de entrada, mantendo a coerência entre diferentes execuções. Essa propriedade é essencial em pesquisas e comparações de algoritmos, pois assegura que a análise de complexidade dependa exclusivamente da estrutura do código e não de fatores externos como compilador, hardware ou tempo de execução (RIVAL; YI, 2020). Em síntese, o uso de uma plataforma web interativa para análise estática de código une teoria, prática e acessibilidade. O sistema permite que qualquer usuário explore o comportamento estrutural de algoritmos, visualize os padrões de complexidade e compreenda as justificativas da classificação Big-O de forma intuitiva. Essa combinação entre parsing automatizado, interpretação formal e interface visual define um novo modelo de aprendizado e pesquisa aplicado à ciência da computação moderna.

Figura 5. Arquitetura de multicamadas de Aplicações Web (PennState, 2022)



RESULTADOS E DISCUSSÃO

Mediante aos resultados alcançados, o projeto prático se deu por meio da utilização de componentes que desempenharam para que a análise estática de algoritmos com geração de complexidade assintótica fosse gerada com sucesso. O projeto final consistiu na implementação de uma ferramenta web interativa, denominada Big O Analyzer, que permite ao usuário inserir um código-fonte em diferentes linguagens de programação e, conforme o código fornecido pelo usuário,

consegue obter uma estimativa da sua complexidade algorítmica no pior caso, acompanhada de uma explicação textual e sugestões de otimização.

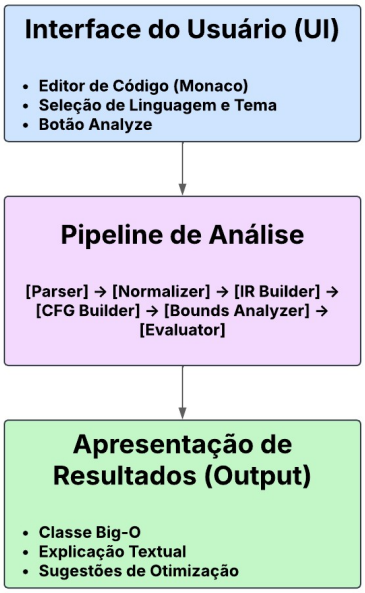
De maneira geral, o sistema foi desenvolvido utilizando o framework Next.js, que possibilitou a construção de uma interface moderna, responsiva e dinâmica. Essa interface, integrada ao editor de código Monaco, permite que o usuário digite, cole ou modifique um trecho de código em linguagens como JavaScript, Python e Java, e visualize os resultados em tempo real. A escolha do Monaco como editor base se justifica por sua capacidade de fornecer recursos avançados, como realce de sintaxe (syntax highlighting), numeração de linhas e detecção automática de erros básicos, características que contribuem significativamente para a experiência do usuário durante a inserção e modificação do código-fonte. O funcionamento da aplicação segue um fluxo de dados modular, em que cada parte do código-fonte é analisada por etapas independentes até a obtenção do resultado.

A arquitetura geral do sistema pode ser representada por um modelo conceitual, conforme representada na Figura 6. Nessa arquitetura, o processo inicia-se na interface do usuário (UI), onde o código é inserido e a linguagem é selecionada. Em seguida, os dados são encaminhados para o pipeline de análise, que executa uma sequência de módulos: AST Parser, Normalizer, IR builder, CFG Builder, Bounds Analyzer e Evaluator. O resultado dessa cadeia de etapas é então devolvido para a interface, que o apresenta em formato de texto, com ícones, cores e dicas visuais de complexidade. O tempo de resposta do sistema foi otimizado para executar em poucos segundos na maioria dos casos de uso testados, garantindo uma experiência fluida e responsiva.

Interface do Usuário

A interface do usuário foi desenvolvida para tornar essa experiência intuitiva e visualmente clara. Ao clicar em Analisar, o sistema exibe um pequeno indicador de carregamento enquanto o pipeline processa as informações. Assim que o resultado é retornado, o usuário visualiza uma janela indicando a classe de complexidade de pior caso, acompanhada da explicação textual e de uma lista de sugestões. As cores dos ícones seguem uma escala intuitiva, onde, verde para O(1), azul para O(log n), amarelo para O(n), laranja para O(n log n) e vermelho para O(n^2) ou de maiores complexidades, auxiliando na rápida identificação da eficiência do código.

Figura 6. Arquitetura geral do sistema Big O Analyzer.



Pipeline de Análise

O pipeline de análise é o núcleo lógico da aplicação. Ele tem como principal função transformar o código-fonte digitado em um conjunto de informações estruturadas que possam ser interpretadas e classificadas. Para isso, o processo ocorre em seis etapas principais, observados na Figura 7, cada uma dessas etapas foi cuidadosamente projetada para operar de forma eficiente e independente, permitindo que o sistema escale adequadamente mesmo com códigos mais complexos ou extensos.

Na primeira etapa, chamada parser, o sistema converte o texto digitado em uma árvore sintática abstrata (AST). Essa árvore é uma representação hierárquica das estruturas do código, como funções, laços e condicionais. O parser foi projetado de forma simples e rápida, priorizando portabilidade entre linguagens. Por exemplo, ele é capaz de identificar um laço for ou while sem precisar compreender todos os detalhes da sintaxe da linguagem. Esse formato permite que o sistema seja facilmente adaptado para outras linguagens de programação no futuro, mantendo o desempenho em tempo real.

A implementação do parser utiliza técnicas de análise léxica baseadas em tokens, onde cada palavra-chave, operador ou identificador é reconhecido e classificado antes da construção da árvore. Essa abordagem, embora simplificada em comparação com parsers completos de compiladores, mostrou-se suficiente para os propósitos educacionais e exploratórios do projeto, mantendo um equilíbrio entre precisão e desempenho.

Após essa conversão inicial, a etapa de normalização transforma os elementos específicos de cada linguagem em um formato padronizado, chamado IR. Essa camada funciona como um “idioma comum” entre diferentes linguagens, garantindo que uma função em Python e uma função equivalente em JavaScript sejam tratadas da mesma forma nas etapas seguintes. A normalização gera uma lista de nós com informações básicas, como o nome da função, tipo de laço e nível de aninhamento.

Durante o processo de normalização, também são eliminadas construções sintáticas irrelevantes para a análise de complexidade, como comentários, strings literais e formatação específica de cada linguagem, focando exclusivamente nas estruturas de controle de fluxo que impactam o desempenho algorítmico. Essa etapa é fundamental para garantir a consistência dos resultados independentemente da linguagem de programação utilizada.

Em seguida, entra em ação o módulo de construção de grafo IR, que organiza esses nós em uma estrutura gráfica composta por pontos de entrada e saída. Essa estrutura, chamada Control Flow Graph (CFG), modela o caminho lógico percorrido pela execução do algoritmo. Por exemplo, cada condição if ou else representa uma bifurcação, e cada laço adiciona uma ligação de retorno, simulando a repetição das instruções, tal trecho é representado nas imagens 8 e 9 respectivamente.

Após o grafo ser montado, o módulo bounds analyzer realiza uma varredura sobre o código-fonte bruto para identificar laços, chamadas recursivas e padrões de divisão de problemas, como ocorre em algoritmos de busca binária ou de ordenação por mesclagem (merge sort). Essa etapa utiliza expressões regulares e heurísticas simples para determinar quantas vezes um bloco de instruções pode ser repetido ou dividido, gerando dados sobre a profundidade de aninhamento e sobre a presença de recursividade.

Figura 7. Representação do fluxo da Pipeline no código.

```
export class AnalysisPipeline {
  private normalizer = new ASTNormalizer();
  private irBuilder = new IRBuilder();
  private cfgBuilder = new CFGBuilder();
  private boundsAnalyzer = new BoundsAnalyzer();
  private evaluator = new ComplexityEvaluator();

  async analyze(code: string, language: Language): Promise<Analysis> {
    await new Promise((resolve) => setTimeout(resolve, 500));
    try {
      // step 1: Parse code into AST
      const ast = ParserFactory.parse(code, language);
      // step 2: Normalize AST into unified IR nodes
      const irNodes = this.normalizer.normalize(ast);
      // step 3: Build Intermediate Representation
      const ir = this.irBuilder.build(irNodes);
      // step 4: Construct Control Flow Graph
      const cfg = this.cfgBuilder.buildCFG(ir);
      // step 5: Analyze bounds (loops and recursion)
      const bounds = this.boundsAnalyzer.analyze(code, language);
      // step 6: Evaluate complexity
      const complexity = this.evaluator.evaluate(code, bounds, language);

      const explanation = COMPLEXITY_EXPLANATIONS[complexity];
      const suggestions = OPTIMIZATION_SUGGESTIONS[complexity];

      return {
        complexity,
        explanation,
        suggestions,
      };
    } catch (error) {
      console.error("[Pipeline] Analysis failed:", error);

      return {
        complexity: "O(n)",
        explanation:
          "Unable to complete analysis. Please check your code syntax.",
        suggestions: ["Ensure your code is syntactically correct."],
      };
    }
  }
}
```

Com base nessas informações, o evaluator (avaliador) combina os resultados das análises anteriores para determinar a classe de complexidade Big O. Ele utiliza regras pré-definidas e um conjunto de constantes armazenadas em uma biblioteca interna (lib/constants). Por exemplo, se o código apresenta um único laço simples, o sistema interpreta a complexidade como O(n). Se há dois laços aninhados, ela é classificada como $O(n^2)$. Se o padrão detectado corresponde a uma divisão recursiva do problema, como em uma busca binária, o resultado é O(log n) ou O(n log n).

O resultado dessa avaliação é retornado para a interface na forma de um objeto contendo três campos principais: complexity, explanation e suggestions. O primeiro campo traz a classe Big-O propriamente dita (como O(1), O(n), O(n log n), $O(n^2)$ etc.); o segundo apresenta uma explicação em linguagem natural, descrevendo o comportamento do algoritmo; e o terceiro oferece sugestões de melhoria, como reduzir laços aninhados ou utilizar métodos nativos mais eficientes.

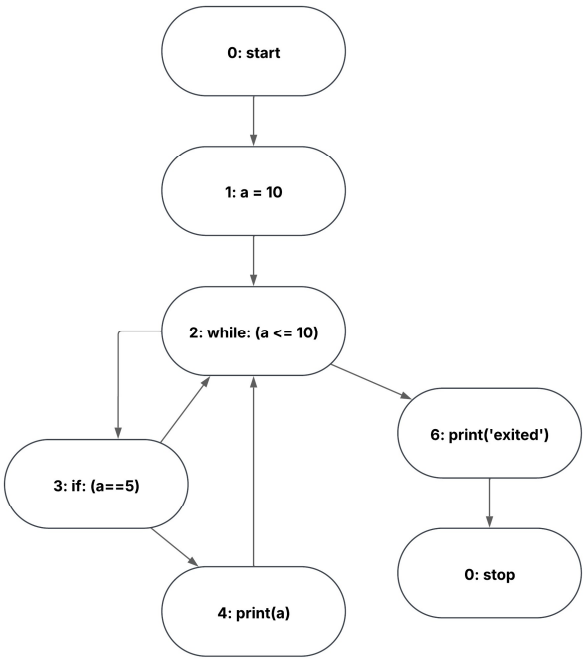
Figura 8. Código representativo em Java para a construção do CFG.

```java
public class Main {
    public static void main(String[] args) {
        int a = 10;

        while (a <= 0) {
            if (a == 5) {
                System.out.println(a);
            }
            a += 1;
        }

        System.out.println("exited");
    }
}
```

Figura 9. Fluxograma do CFG em Java.



Apresentação de Resultados

Como exemplo de funcionamento, foi testado um trecho clássico de código em Python com base na tabela 2, referente ao algoritmo de ordenação por bolha (Bubble Sort), reproduzido na figura 10. O sistema analisou corretamente a presença de dois laços aninhados, identificando que o número de comparações cresce proporcionalmente ao quadrado do tamanho da entrada. Dessa forma, classificou o algoritmo como pertencente à classe $O(n^2)$. Além disso, apresentou uma explicação descritiva e uma sugestão de otimização, ressaltando que, embora o código seja funcional, sua eficiência é limitada para grandes volumes de dados, podendo ser substituído por métodos mais eficientes, como Merge Sort ou Quick Sort, representada na figura 11.

Esse resultado evidencia que o sistema cumpre seu objetivo principal: tornar acessível o entendimento da eficiência de algoritmos de maneira automática e pedagógica. O Big O Analyzer não pretende substituir ferramentas de análise formal de compiladores, mas oferecer uma abordagem educacional simplificada, voltada para estudantes e desenvolvedores que desejam compreender, de forma heurística, o impacto de suas decisões lógicas no desempenho do código.

Durante a implementação, buscou-se equilibrar a velocidade da análise e a precisão dos resultados. O uso de parsers simplificados reduziu a carga de processamento e aumentou a compatibilidade entre linguagens, ainda que isso signifique renunciar a algumas minúcias sintáticas. Na prática, esse compromisso mostrou-se vantajoso: o sistema é capaz de processar trechos de código em milissegundos, mantendo a fluidez da interface, mesmo em navegadores comuns.

Outro ponto relevante foi a utilização de constantes centralizadas, que padronizam palavras-chave, estruturas de controle e métodos nativos de bibliotecas comuns. Essa escolha facilitou a manutenção e ampliou a escalabilidade do projeto. A adição de novas linguagens, por exemplo, exige apenas a criação de um novo parser leve e o registro de suas palavras-chave correspondentes.

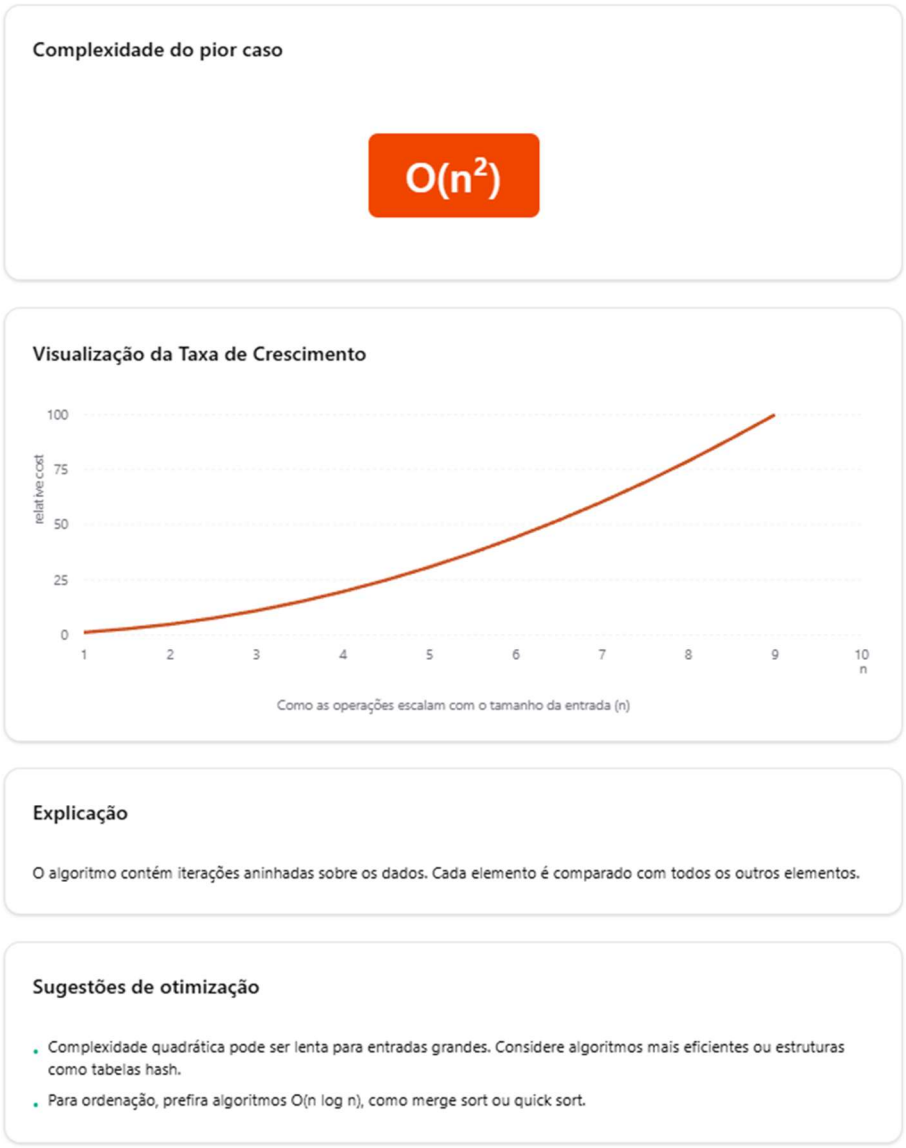
Tabela 2. Evidências da Complexidade Assintótica dos Algoritmos Testados pelo Big O Analyzer

Algoritmo	**Linguagem**	**Complexidade teórica (pior caso)**	**Complexidade indicada pelo sistema**	**Resultado**	**Observações rápidas**
Busca linear em vetor	Python	O(n)	O(n)	Correto	Percorre todos os elementos do vetor uma vez; tempo cresce linearmente com n.
Busca binária em vetor ordenado	Java	O(log n)	O(log n)	Correto	A cada passo o espaço de busca é reduzido pela metade.
Subset Sum (busca exaustiva de subconjuntos)	JavaScript	$O(2^n)$	$O(2^n)$	Correto	Percorre todos os subconjuntos possíveis do conjunto de n elementos.
Ordenação por bolha (Bubble Sort)	Python	$O(n^2)$	$O(n^2)$	Correto	Dois laços aninhados com comparações sucessivas entre pares de elementos.
Ordenação por mesclagem (Merge Sort)	JavaScript	O(n log n)	O(n log n)	Correto	Divide o problema em sublistas recursivamente e faz mesclagem linear em cada nível.

Figura 10. Código em Python (Bubble Sort).

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

Figura 11. Resultados demonstrado pelo website após a análise do código em Python.



Entre os desafios enfrentados durante o desenvolvimento, destacam-se as limitações inerentes à análise heurística. Como o sistema não executa o código de fato, ele depende da identificação de padrões estruturais. Isso significa que certos algoritmos que utilizam metaprogramação, geração dinâmica de funções ou bibliotecas externas podem não ter suas complexidades corretamente inferidas. Entretanto, essa limitação não compromete o propósito principal do projeto, que é o uso em contextos educacionais e exploratórios.

Em termos de resultados, o Big O Analyzer demonstrou consistência e clareza na classificação

de algoritmos comuns, como busca linear, ordenação por inserção, busca binária e ordenação por mesclagem. A interface amigável e as explicações em linguagem natural tornaram o sistema útil para introduzir estudantes aos conceitos de complexidade assintótica e eficiência de algoritmos, de modo visual e interativo.

Dessa forma, pode-se afirmar que o projeto atendeu integralmente aos objetivos propostos. A integração entre os módulos, a fluidez do pipeline e a simplicidade da interface reforçam o potencial da aplicação tanto como ferramenta didática quanto como base para evoluções futuras. Caso sejam incorporadas novas linguagens e aprimorados os detectores de recursividade, o sistema poderá oferecer resultados ainda mais completos e precisos.

Em síntese, o desenvolvimento do Big O Analyzer demonstrou que é possível construir uma solução prática, acessível e pedagógica para análise estática de algoritmos, aliando conceitos teóricos de complexidade assintótica a uma experiência interativa e intuitiva.

CONSIDERAÇÕES FINAIS

O desenvolvimento deste trabalho demonstrou que é possível realizar a análise estática de algoritmos com geração automática de complexidade assintótica, atendendo plenamente ao objetivo proposto na introdução. A criação do sistema Big O Analyzer validou a hipótese de que heurísticas estruturais baseadas em AST e CFG podem ser aplicadas de forma prática para estimar o comportamento assintótico de diferentes algoritmos sem necessidade de execução do código.

A ferramenta desenvolvida mostrou-se eficaz ao identificar corretamente padrões de loops, recursões e estruturas de controle, classificando-os em classes de complexidade Big-O com explicações claras e acessíveis. O projeto também evidenciou a relevância da integração entre teoria e prática: conceitos tradicionalmente abstratos da análise de algoritmos foram traduzidos em representações visuais e interpretações automáticas que facilitam o aprendizado e a compreensão de estudantes e desenvolvedores.

A abordagem adotada conciliou eficiência técnica e simplicidade pedagógica, permitindo respostas rápidas e uma experiência interativa em ambiente web. Além disso, o uso de tecnologias modernas como Next.js e Monaco Editor proporcionou uma interface responsiva e intuitiva, reforçando o caráter acessível e didático da solução.

Entre as principais contribuições do trabalho estão a demonstração da viabilidade de se realizar inferências assintóticas sem execução do código, a padronização do processo de análise por meio de normalização entre linguagens e a proposta de um modelo visual de apoio à aprendizagem. Embora o sistema apresente limitações em casos que evolvem metaprogramação ou dependência de bibliotecas externas, ele cumpre seu propósito principal de forma satisfatória e abre caminhos para aprimoramentos futuros.

Como próximos passos, propõe-se a expansão do Big O Analyzer em uma frente primordialmente técnica. O sistema pode ser evoluído com parsers mais robustos, capazes de lidar com diferentes estilos de escrita de código, aliado ao suporte a um conjunto mais amplo de linguagens de programação. Além disso, a incorporação de heurísticas mais sofisticadas permitirá tratar com maior precisão cenários complexos, como o uso intensivo de bibliotecas e casos de recursão indireta, ampliando tanto a confiabilidade quanto o alcance prático da ferramenta.

REFERÊNCIAS BIBLIOGRÁFICAS

ADAMKÓ, Attila. Internet Tools and Services: Layered Architecture for Web Applications. [S. l.], 2014. University of Debrecen. Disponível em: https://gyires.inf.unideb.hu/GyBITT/08/index.html. Acesso em: 29 out. 2025.

ALBERT, Elvira; HÄHNLE, Reiner; MERAYO, Alicia; STEINHÖFEL, Dominic. Certified Cost Bounds for Abstract Programs. ACM Transactions on Software Engineering and Methodology, v. 34, n. 3, p. 67:1–67:33, 2025. Disponível em: https://dl.acm.org/doi/10.1145/3705298. Acesso em: 2 out. 2025.

AHO, Alfred V. *et al*. Compiladores: Princípios, técnicas e ferramentas. 2. ed. [*S. l.*]: Pearson Education, 2008. Disponível em: https://www.scribd.com/document/706174206/Compiladores-2nd#page=348&content=query%3Agrafo+de+fluxo%2CpageNum%3A348%2CindexOnPage%3A3%2CbestMatch%3Afalse. Acesso em: 19 set. 2025.
BERNERS-LEE, Tim. *Information Management: A Proposal*. Genebra: CERN, 1989. Disponível em: https://cds.cern.ch/record/369245/files/dd-89-001.pdf. Acesso em: 3 out. 2025.
CORMEN, Thomas H. *et al*. Algoritmos: Teoria e prática. 3. ed. [*S. l.*]: MIT Press, 2009.
COUSOT, Patrick; COUSOT, Radhia. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '77)*. New York: ACM, 1977. p. 238–252. Disponível em: https://dl.acm.org/doi/10.1145/512950.512973. Acesso em: 1 out. 2025.
FREE SOFTWARE FOUNDATION. GNU Compiler Collection (GCC) Internals — Control Flow Graph. Disponível em: https://gcc.gnu.org/onlinedocs/gccint/Control-Flow.html. Acesso em: 2 out. 2025.
JETBRAINS. Guia de análise estática de código. [S. l.], 2025. Disponível em: https://www.jetbrains.com/pt-br/pages/static-code-analysis-guide/#why-use-static-code-analysis. Acesso em: 3 out. 2025.
KOSINSKI, Matthew et al. O que é o MapReduce?. In: HOLDSWORTH, Jim et al. O que é o MapReduce? [S. l.], 19 nov. 2024. Disponível em: https://www.ibm.com/br-pt/think/topics/mapreduce. Acesso em: 27 out. 2025.
LLVM PROJECT. LLVM Language Reference Manual. 2018. Disponível em: https://buildmedia.readthedocs.org/media/pdf/llvm/latest/llvm.pdf. Acesso em: 1 out. 2025.
META OPEN SOURCE. React. Disponível em: https://react.dev/. Acesso em: 23 set. 2025.
META OPEN SOURCE. Rules of React. Disponível em: https://react.dev/reference/rules. Acesso em: 23 set. 2025.
MICROSOFT. Monaco Editor. Disponível em: https://microsoft.github.io/monaco-editor/. Acesso em: 24 set. 2025.
NIELSON, Flemming et al. Principles of Program Analysis. [S. l.]: Springer, 1998. Disponível em: https://www.researchgate.net/publication/265352570_Principles_of_Program_Analysis. Acesso em: 28 out. 2025.
ON the reliability of mechanisms. In: DIJKSTRA, Edsger W. NOTES ON STRUCTURED PROGRAMMING. 2. ed. [S. l.]: T.H. - Report 70-WSK-03, 1970. Disponível em: https://www.cs.utexas.edu/~EWD/transcriptions/EWD02xx/EWD249/EWD249.html. Acesso em: 6 set. 2025.
SEDGEWICK, Robert; WAYNE, Kevin. Algorithms: FOURTH EDITION. 4. ed. [S. l.]: Addison-Wesley, 2011. Disponível em: https://mrce.in/ebooks/Algorithms%204th%20Ed.pdf. Acesso em: 19 set. 2025.
SIPSER, Michael. Introduction to the Theory of Computation. 3. ed. [*S. l.*]: Cengage Learning, 2013. Disponível em: https://cs.brown.edu/courses/csci1810/fall-2023/resources/ch2_readings/Sipser_Introduction.to.the.Theory.of.Computation.3E.pdf. Acesso em: 18 set. 2025.
THE O Notation. In: KNUTH, Donald E. The Art of Computer Programming. 3. ed. [S. l.]: Addison-Wesley Professional, 1997. v. 1, cap. 22, p. 107-110. Disponível em: https://seriouscomputerist.atariverse.com/media/pdf/book/Art%20of%20Computer%20Programming%20-%20Volume%201%20(Fundamental%20Algorithms).pdf. Acesso em: 6 set. 2025.
VERCEL. Getting Started: Route Handlers and Middleware. Disponível em: https://nextjs.org/docs/app/getting-started/route-handlers-and-middleware. Acesso em: 30 set. 2025.
VERCEL. Next.js App Router (Docs). Disponível em: https://nextjs.org/docs/app. Acesso em: 30 set. 2025.
VERCEL. Rendering: Server Components. Disponível em: https://nextjs.org/docs/14/app/building-your-application/rendering/server-components. Acesso em: 30 set. 2025.

WORLD WIDE WEB CONSORTIUM (W3C). *HTML5: A vocabulary and associated APIs for HTML and XHTML*. W3C Recommendation, 28 out. 2014. Disponível em: https://www.w3.org/TR/2014/REC-html5-20141028/. Acesso em: 3 out. 2025.

AGRADECIMENTOS

Gostaríamos de manifestar nossos mais sinceros agradecimentos a todos que contribuíram para a realização deste projeto. Em especial, ao nosso orientador e aos professores Prof. Me. Clayton Valdo, Fernando Domeneghetti e Tetsuo Araki, que foram fundamentais para o nosso desenvolvimento ao longo do curso. Agradecemos também às nossas famílias, por sempre nos encorajarem, apoiarem e oferecerem todo o suporte necessário em cada etapa dessa jornada. Por fim, expressamos nossa gratidão aos amigos que fizemos e levaremos conosco, bem como a todas as pessoas que, de forma direta ou indireta, colaboraram para a concretização deste trabalho.