Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Análise Dinâmica de Código a Partir de Representação Visual em Tempo Real Usando Three.js

João Paulo de Carvalho Araújo

JUIZ DE FORA
JUNHO, 2024

Análise Dinâmica de Código a Partir de Representação Visual em Tempo Real Usando Three.js

João Paulo de Carvalho Araújo

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Ciro de Barros Barbosa

JUIZ DE FORA

JUNHO, 2024

Análise Dinâmica de Código a Partir de Representação Visual em Tempo Real Usando Three.js

João Paulo de Carvalho Araújo

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Ciro de Barros Barbosa
Doutor em Ciência da Computação

André Luiz de Oliveira
Doutor em Ciências da Computação e Matemática Computacional

Rodrigo Luis de Souza da Silva
Doutor em Engenharia Civil

JUIZ DE FORA
22 DE JUNHO, 2024

À minha esposa.

À minha mãe, pelo apoio e sustento.

Resumo

Este trabalho propõe a análise dinâmica de código utilizando representações visuais em tempo real com a biblioteca *Three.js*, visando facilitar o entendimento de estruturas de dados e algoritmos para estudantes e profissionais de Ciência da Computação. A *API* desenvolvida permite a visualização dinâmica do comportamento de estruturas como pilhas, filas e árvores, complementando o rol de ferramentas que auxiliem na compreensão dessas operações. Por se tratar de uma aplicação externa, pode ser integrada à quaisquer linguagens de programação capazes de realizar requisições *HTTP* ao servidor, que gerencia o modelo a ser apresentado no navegador *web*.

Palavras-chave: depuração, análise dinâmica, visualização interativa, *API REST*.

Abstract

This work proposes dynamic code analysis using real-time visual representations with the Three.js library, aiming to facilitate the understanding of data structures and algorithms for students and professionals in Computer Science. The developed API allows dynamic visualization of the behavior of structures such as stacks, queues, and trees, complementing the range of tools that aid in understanding these operations. As an external application, it can be integrated with any programming languages capable of making HTTP requests to the server, which manages the model to be presented in the web browser.

Keywords: debugging, dynamic analysis, interactive visualization, REST API.

Agradecimentos

À minha esposa e à minha mãe, pelo encorajamento e apoio.

Ao professor Ciro Barbosa pela orientação e principalmente, pela paciência, sem a qual este trabalho não se realizaria.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos e aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

"O homem não teria alcançado o possível se, repetidas vezes, não tivesse tentado o impossível".

Max Weber

Conteúdo

Lista de Figuras

Lista de Abreviações

IDE	Integrated Development Environment
ORM	Object-Relational Mapping
SGBD	Sistema Gerenciador de Banco de Dados
DSA	Data Structures and Algorithms
GFC	Grafo de Fluxo de Controle
DFT	Depth-First Traversal

1 Introdução

A análise dinâmica de código é um processo fundamental no desenvolvimento de *software* que envolve a execução do programa para observar seu comportamento em tempo de execução. Essa abordagem permite coletar informações detalhadas sobre como o *software* interage com o sistema operacional, recursos de hardware e dados de entrada.

Segundo Huang et al. (2015), a análise dinâmica é essencial para identificar problemas que podem não ser facilmente detectados por meio de análises estáticas, como a detecção de vulnerabilidades de segurança, comportamentos inesperados e otimização de desempenho. Além disso, a análise dinâmica é crucial para compreender o comportamento real de um programa em diferentes cenários de uso, fornecendo *insights* valiosos sobre as partes do código que estão sendo executadas e identificando áreas que precisam de mais atenção durante o desenvolvimento e teste do *software*.

1.1 Contextualização

A visualização de estruturas de dados é uma ferramenta essencial no desenvolvimento de *software*, pois permite aos desenvolvedores compreender de forma mais clara e interativa como os dados são organizados e manipulados em um programa de computador. A representação gráfica das estruturas de dados facilita a visualização de como os elementos estão interconectados e como as operações são realizadas sobre esses dados (LIN; ZHANG, 2020).

A visualização dinâmica em tempo real possibilita acompanhar as mudanças nas estruturas de dados à medida que o programa é executado, tornando mais fácil identificar possíveis problemas e otimizar o desempenho do código. A interatividade proporcionada pela visualização permite aos desenvolvedores realizar operações como inserção, remoção e atualização de elementos, contribuindo para um melhor entendimento do funcionamento das estruturas de dados e auxiliando no processo de aprendizado.

A visualização de estruturas de dados em tempo real, combinada com a análise

dinâmica de código, oferece uma abordagem abrangente e eficaz para o desenvolvimento de *software* de qualidade, permitindo uma compreensão mais profunda do comportamento do programa e facilitando a identificação de possíveis falhas e melhorias.

1.2 Descrição do problema

Programas de computador são formados por estruturas abstratas, que são difíceis de visualizar mentalmente. Na programação orientada a objetos, a execução pode gerar centenas de objetos interagindo entre si. No ensino de algoritmos e estruturas de dados, alunos podem ter dificuldades em compreender a dinâmica dos dados e a relação entre eles ao longo da execução do código.

Avaliar a complexidade de um determinado sistema também é uma análise importante, visto que novos desenvolvedores precisam compreender a arquitetura de uma aplicação rapidamente, visando dar continuidade ao trabalho de outros programadores.

1.3 Justificativa

A crescente complexidade do desenvolvimento de *software* e a demanda por soluções rápidas e eficientes tornam imperativa a adoção de métodos de ensino que proporcionem uma compreensão mais profunda e prática dos conceitos fundamentais da programação. A utilização de representações gráficas e visualizações em tempo real não apenas facilita a absorção do conhecimento, mas também prepara os alunos para enfrentar os desafios práticos que encontrarão em projetos futuros.

Além disso, a implementação de uma *API* que permita a depuração e verificação de algoritmos em tempo real oferece um suporte valioso para o aprendizado, capacitando os estudantes a identificar e corrigir erros de maneira mais eficaz. Essa abordagem empodera os alunos a se tornarem desenvolvedores mais competentes, contribuindo para a melhoria da qualidade do *software* produzido e a eficiência do processo de desenvolvimento como um todo.

1.4 Objetivos

O objetivo principal deste trabalho é desenvolver uma API que permita a visualização dinâmica do comportamento de estruturas de dados por meio de representações visuais em tempo real. Para alcançar este objetivo são considerados os seguintes objetivos específicos:

- Desenvolvimento de uma aplicação que gerencie as informações relacionadas ao código ou às estruturas de dados utilizadas;
- A criação de uma interface web que represente graficamente os dados enviados;
- Implementar uma classe que encapsule as requisições feitas à aplicação, facilitando a integração entre código-fonte e *backend* remoto;
- Criar código-fonte de teste, que utilize a classe acima para validação do projeto.

2 Fundamentação Teórica

2.1 Teste de *Software*

Segundo Delamaro, Maldonado e Jino (2007), o teste é um processo fundamental no desenvolvimento de *software*. Ele envolve a execução do programa com o objetivo de revelar a presença de erros. Os testes podem ser automatizados e são realizados com ferramentas específicas. A preocupação principal durante os testes é com o resultado obtido, não com o processo de como se chegou a ele. Em outras palavras, os testes são feitos para identificar quando um código está correto, não quando ele tem problemas. No entanto, é importante lembrar que passar nos testes não garante a ausência de *bugs*; apenas indica que certas situações aconteceram corretamente.

A literatura define alguns termos importantes, tais como:

- Defeito: Um defeito é uma imperfeição ou inconsistência no produto de *software* ou em seu processo. Pode estar presente no código, na documentação ou em outras partes do *software*. Um defeito faz parte do produto e é algo que está implementado de forma incorreta;

- Erro: Um erro é um estado de execução inconsistente. Pode ser resultado de um defeito ou de uma falha. Por exemplo, um retorno esperado que, devido a uma falha, tem um valor diferente do esperado;

- Falha: Uma falha é um evento notável em que o sistema viola suas especificações. Está mais relacionada ao hardware, como uma rede inacessível ou queda de energia. Uma falha pode ocorrer devido a um erro, como um retorno de valor não esperado (por exemplo, null), que afeta o funcionamento do sistema;

- Engano: Refere-se à ação humana que resulta na introdução de um defeito no código ou em algum outro produto de trabalho relacionado. Esses erros podem ocorrer durante o processo de desenvolvimento, como quando um desenvolvedor comete um

equívoco ao programar. O engano é uma parte inerente do ciclo de desenvolvimento de *software* e destaca a importância de práticas rigorosas de revisão, testes e controle de qualidade para minimizar sua ocorrência. Quando os enganos não são identificados e corrigidos, podem levar a falhas no sistema, afetando a usabilidade, segurança e confiabilidade do *software* (SULLIVAN; CHILLAREGE, 1991).

2.1.1 Verificação de *Software*

A verificação de *software* envolve uma série de técnicas e práticas que têm como objetivo garantir que o *software* esteja em conformidade com suas especificações e requisitos antes de ser lançado. As atividades de Validação, Verificação e Teste (VV&T) são essenciais nesse processo e não se restringem apenas ao produto final; deve-se conduzi-las durante todo o desenvolvimento do *software*, desde a sua concepção (DELAMARO; MALDONADO; JINO, 2007).

Dentre as abordagens de teste, destacam-se as seguintes fases:

1. Teste de Unidade: Focado nas menores unidades do programa, como funções ou métodos. Esse teste pode ser aplicado enquanto as unidades estão sendo implementadas, permitindo que erros em funcionalidades específicas sejam identificados precocemente.

2. Teste de Integração: Realizado após o teste das unidades, seu objetivo é verificar a interação entre diferentes partes do *software* e assegurar que elas funcionem em conjunto sem causar erros.

3. Teste de Sistemas: Esta fase examina o sistema como um todo para garantir que os requisitos do *software* são atendidos.

Além disso, as técnicas estruturais, como a aplicação de critérios de teste, são consideradas relevantes para atividades de manutenção e avaliação da confiabilidade do *software*. A confiabilidade, por sua vez, é uma métrica importante que reflete a qualidade do *software* do ponto de vista do usuário e está sujeita ao grau de defeitos remanescentes.

2.1.2 Técnica de Teste Estrutural

A técnica de teste estrutural, também conhecida como teste de caixa branca, é uma abordagem que define os requisitos de teste com base na implementação do *software*. Essa técnica requer que sejam realizadas execuções de partes ou componentes do código, permitindo que os testadores examinem a estrutura interna do programa ao invés de somente suas saídas (DELAMARO; MALDONADO; JINO, 2007).

Os critérios de teste estruturais são variados e incluem aspectos como:

- Critérios de Fluxo de Controle: Focados em testar a lógica de execução do código, garantindo que todas as ramificações sejam percorridas pelo menos uma vez.
- Critérios de Fluxo de Dados: Estes critérios baseiam-se na análise das interações entre as definições de variáveis e suas referências subsequentes, o que é fundamental para garantir que as variáveis são utilizadas corretamente.
- Teste de Mutação: Uma técnica que altera o código para criar versões chamadas "mutantes"e os utiliza para verificar se os testes existentes conseguem detectar as alterações, ajudando assim a medir a eficácia dos casos de testes criados.

A correta escolha e aplicação dessas técnicas pode levar a uma revelação mais eficaz de defeitos, além de auxiliar na construção de uma base de conhecimento sobre a eficácia de cada critério de teste. As abordagens complementares tão comuns em testes estruturais permitem que diferentes tipos de defeitos sejam abordados, enriquecendo o processo de teste como um todo.

Ainda segundo Delamaro, Maldonado e Jino (2007), a implementação de estratégias incrementais, onde critérios "mais fracos"são aplicados inicialmente, pode facilitar a gestão de custos e recursos em ambientes de teste.

2.1.3 Grafo de Fluxo de Controle

O Grafo de Fluxo de Controle (GFC) é uma ferramenta fundamental na técnica de teste estrutural. Ele representa a sequência de execução de um programa, mostrando como o controle flui entre diferentes partes do código. A partir desse grafo, é possível identificar

os caminhos de execução e os pontos críticos que devem ser testados para garantir a qualidade e a funcionalidade do *software* (DELAMARO; MALDONADO; JINO, 2007).

As principais características dos grafos utilizados no teste estrutural são:

- Grafo de Fluxo de Controle (GFC): Este grafo é construído para cada função e contém informações sobre o fluxo de controle do programa. Cada nó do grafo representa um ponto no código, enquanto as arestas indicam as possibilidades de transição entre esses pontos.

- Grafo Def-Uso: Uma extensão do GFC que inclui informações sobre as definições e usos de variáveis. Esse grafo permite aos testadores visualizar as associações entre onde uma variável é definida e onde seu valor é utilizado, o que é crucial para a criação de casos de teste adequados.

- Grafo de Fluxo de Controle Interprocedimental (GFCI): Este grafo abrange o fluxo de controle que atravessa diferentes funções, permitindo a análise de como as chamadas entre funções ocorrem e afetando a execução geral do programa.

- Grafo de Fluxo de Controle Composto (GFCC): Utilizado para representar a interação entre diferentes níveis de controle em um sistema, ajudando na identificação de requisitos de teste interclasse.

Esses grafos não somente ajudam na identificação de caminhos críticos que precisam ser testados, como também são essenciais para a definição de casos de teste que abrangem todos os caminhos e combinações possíveis dentro do código. Assim, eles suportam a montagem de estratégias de teste mais robustas e eficazes.

A figura 2.1 apresenta um exemplo de Grafo de Fluxo de Controle gerado usando a aplicação *View Graph*, onde trechos do código-fonte são agrupados em nós e as estrutruras de controle e repetição definem os desvios no fluxo de execução.

2.1.4 Critérios de Teste

Segundo Delamaro, Maldonado e Jino (2007), os critérios de teste são frameworks fundamentais que orientam o desenvolvimento de casos de teste para garantir a qualidade

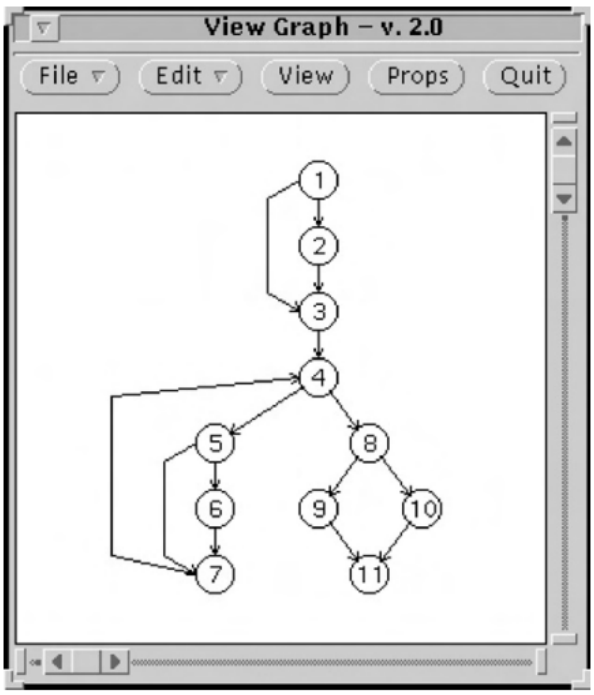

Figura 2.1: Exemplo de Grafo de Fluxo de Controle (DELAMARO; MALDONADO; JINO, 2007)

do *software*. Eles definem regras e diretrizes para garantir que diferentes aspectos do *software* sejam examinados durante o processo de teste. Abaixo estão os principais tipos de critérios de teste, com foco nos critérios estruturais e suas características:

1. Critérios Estruturais: Esses critérios são baseados na estrutura do código e incluem:

   - Todos-Nós (*All-Node*): Este critério requer que todos os nós no grafo de fluxo de controle do programa sejam executados pelo menos uma vez. É o mínimo esperado em uma boa atividade de teste, pois garante que cada comando do programa tenha sido testado.
   - Todas-Arestas (*All-Edges*): Exige que todas as arestas (ou desvios) no grafo de fluxo de controle sejam percorridas. Este critério é essencial para garantir que todas as transições possíveis entre nós sejam testadas.
   - Todos-Caminhos (*All-Paths*): Este critério ideal requer que todos os caminhos possíveis dentro de um programa sejam executados pelo menos uma vez. No entanto, na prática, isso é muitas vezes impraticável devido ao grande número de caminhos que podem existir, especialmente em programas complexos.

2. Critérios Baseados em Fluxo de Dados: Esses critérios focam na interação entre as definições de variáveis e seus usos. Eles são importantes para identificar se as variáveis são corretamente definidas e usadas dentro do seu escopo.

3. Critérios Funcionais: Baseiam-se em especificações funcionais do *software*. Um exemplo típico é o Particionamento de Equivalência, que divide o domínio de entrada do *software* em grupos e exige que pelo menos um valor de cada grupo seja testado, garantindo assim uma cobertura mais abrangente de casos de teste.

Os critérios de teste são essenciais para a validação da funcionalidade e qualidade do *software*, e sua escolha deve ser guiada pela necessidade de balancear a eficácia do teste e os custos associados à sua implementação.

2.1.5 Depuração

Segundo Delamaro, Maldonado e Jino (2007), a depuração de *software* é definida como o processo de localização e remoção de defeitos que surgem durante o desenvolvimento de um programa. Este processo é frequentemente iniciado após a execução de testes que revelam a presença de falhas no *software*. A depuração pode ocorrer em diferentes momentos do ciclo de vida do *software*, sendo as principais fases:

1. Durante a codificação: Neste estágio, a depuração complementa a própria programação, permitindo que os desenvolvedores identifiquem e corrijam erros enquanto ainda estão implementando o código.

2. Após o teste: Esta fase é ativada quando os testes revelam defeitos. A depuração beneficia-se das informações coletadas durante os testes, ajudando a localizar as causas das falhas e sugerindo como corrigi-las.

3. Durante a manutenção: Aqui, a depuração busca resolver problemas que podem ter surgido após a liberação do *software*, seja por defeitos descobertos ou pela necessidade de adicionar novas características.

A atividade de depuração se beneficia da utilização de técnicas sistemáticas mais modernas e ferramentas que utilizam informação de teste. Apesar de muitas vezes ainda

depender da habilidade e experiência dos depuradores, esforços recentes têm sido direcionados para aumentar a eficácia e a produtividade desse processo.

Além disso, problemas de indecidibilidade podem dificultar a geração automática de dados de teste, o que representa um obstáculo à automatização completa desse processo. A depuração, portanto, se torna uma parte crítica do ciclo de desenvolvimento de *software*, influenciando a qualidade e confiabilidade do produto final(DELAMARO; MALDONADO; JINO, 2007).

2.1.6 Análise Estática de Código

Segundo Ashfaq, Khan e Farooq (2019), a análise estática de código é um processo de verificação do código-fonte de um programa sem executá-lo. Ela é realizada por ferramentas automatizadas que examinam o código em busca de possíveis erros, violações de padrões de codificação, vulnerabilidades de segurança e más práticas de programação. Essa análise é feita com base nas regras definidas pela equipe de desenvolvimento ou por padrões de codificação reconhecidos.

As ferramentas de análise estática de código são amplamente utilizadas no desenvolvimento de *software* para garantir a qualidade do código e facilitar a identificação de problemas antes mesmo da execução do programa. Essas ferramentas ajudam a melhorar a consistência, legibilidade e manutenibilidade do código, além de contribuir para a detecção precoce de possíveis falhas.

Ao integrar esta etapa ao ciclo de desenvolvimento de *software*, as equipes podem identificar e corrigir problemas de forma mais eficiente, reduzindo custos e melhorando a qualidade do produto final. Além disso, esta é uma prática recomendada para garantir a conformidade com os padrões de codificação e as melhores práticas da indústria de desenvolvimento de *software*.

Existem várias ferramentas disponíveis, cada uma com suas próprias características e funcionalidades. É importante escolher a mais adequada com base nas necessidades do projeto e nos padrões de codificação a serem seguidos. Algumas das ferramentas populares incluem o *Checkstyle*, *PMD*, *FindBugs* e *SonarQube* (ASHFAQ; KHAN; FAROOQ, 2019).

2.1.7 Análise Dinâmica de Código

A análise dinâmica de código é um processo de avaliação de *software* que envolve a execução do programa para observar seu comportamento em tempo de execução. Nesse contexto, ferramentas de análise dinâmica coletam informações detalhadas sobre como o *software* interage com o sistema operacional, recursos de hardware e dados de entrada. Isso permite identificar possíveis vulnerabilidades, *bugs*, comportamentos maliciosos e até mesmo medir a cobertura de código executado durante a execução do programa (HUANG et al., 2015).

Esta é especialmente útil para identificar problemas que podem não ser facilmente detectados por meio de análises estáticas, como a detecção de vulnerabilidades de segurança, comportamentos inesperados e otimização de desempenho. Além disso, a análise dinâmica pode ser essencial para compreender o comportamento real de um programa em diferentes cenários de uso.

No contexto de aplicativos Android, por exemplo, a análise dinâmica de código é fundamental para avaliar a segurança e a qualidade do *software*, especialmente devido à diversidade de dispositivos e ambientes nos quais os aplicativos podem ser executados. Segundo Huang et al. (2015), medir a cobertura de código durante a análise dinâmica pode fornecer *insights* valiosos sobre quais partes do código estão sendo executadas e ajudar a identificar áreas que precisam de mais atenção durante o processo de desenvolvimento e teste.

2.2 Visualização de Estruturas de Dados

2.2.1 Conceito

A visualização de estruturas de dados é uma ferramenta essencial para facilitar a compreensão de como os dados são organizados e manipulados em programas de computador. No contexto do artigo *Data Structure Visualization on the Web* (LIN; ZHANG, 2020), os autores propõem uma ferramenta de visualização de estruturas de dados online que permite expressar várias estruturas de dados com suas representações gráficas.

Essa ferramenta é baseada em Python e é capaz de representar visualmente as

estruturas de dados abstratas, como pilhas, filas, listas encadeadas, entre outras. A visualização dessas estruturas de dados por meio de gráficos e diagramas ajuda os usuários a compreender não apenas a estrutura de dados em si, mas também a dinâmica dos elementos de dados durante a execução do programa.

Além disso, a visualização de estruturas de dados em tempo real e de forma dinâmica é fundamental para capturar e ilustrar não apenas a estrutura estática dos dados, mas também a dinâmica da coleção de nós de dados conforme o programa é executado, o que auxilia estudantes a entender os tipos de dados abstratos e aplicar as implementações apropriadas (LIN; ZHANG, 2020).

Essa abordagem de visualização de estruturas de dados não só torna o aprendizado mais acessível e interativo, mas também ajuda os usuários a compreender melhor a lógica por trás das operações em diferentes estruturas de dados, como pilhas, filas, listas encadeadas, árvores, entre outras.

2.2.2 Visualização interativa

A visualização interativa de estruturas de dados é uma funcionalidade essencial em sistemas educacionais que abordam algoritmos e estruturas de dados. Essa abordagem permite aos alunos visualizarem o funcionamento interno das estruturas de dados de forma dinâmica e interativa, o que facilita a compreensão de conceitos complexos (PERHÁC; SIMONÁK, 2022).

Algumas características comuns incluem:

- Passo a passo: Os alunos podem avançar e retroceder nas operações realizadas na estrutura de dados, permitindo uma visualização detalhada de cada etapa.
- Controles de reprodução: Funcionalidades como pausar, reproduzir e ajustar a velocidade da visualização ajudam os alunos a acompanhar o processo de forma adequada.
- Interatividade: Os alunos podem interagir com a estrutura de dados, realizando operações como inserção, remoção e atualização de elementos.

- Destaque de elementos: Destacar elementos específicos da estrutura de dados durante a visualização ajuda a enfatizar pontos importantes.

- Explicações e pseudocódigo: Fornecer explicações claras sobre as operações realizadas e apresentar o pseudocódigo correspondente auxilia na compreensão dos algoritmos associados às estruturas de dados.

Essa abordagem de visualização interativa de estruturas de dados é amplamente utilizada em ambientes educacionais para tornar o aprendizado de algoritmos e estruturas de dados mais eficaz e envolvente.

Em *Interactive System for Algorithm and Data Structure Visualization* Perhác e Simonák (2022), o sistema desenvolvido apresenta diversas funcionalidades para auxiliar no aprendizado de algoritmos e estruturas de dados. Algumas das funcionalidades incluem:

- Visualizações interativas de algoritmos e estruturas de dados.

- Possibilidade de avançar e retroceder nas visualizações passo a passo.

- Opção de pausar e reproduzir as visualizações.

- Controle da velocidade de reprodução das visualizações.

- Interatividade com as estruturas de dados visualizadas.

- Disponibilidade de visualizações pré-definidas para estruturas de dados básicas.

- Exercícios interativos para testar o entendimento dos algoritmos visualizados.

- Facilidade de extensão do sistema com novas visualizações.

- Suporte a pseudocódigo para auxiliar na compreensão dos algoritmos.

- Explicações breves sobre os algoritmos e estruturas de dados visualizados.

Essas funcionalidades visam tornar o processo de aprendizado mais dinâmico, interativo e compreensível para os estudantes de algoritmos e estruturas de dados.

3 Revisão Bibliográfica

Este capítulo explora artigos acadêmicos, que visam utilizar representações visuais para auxiliar na compreenção de temas ligados a algorimos e estruturas de dados. Ao final de cada sessão é apresentado o contexto em que o presente trabalho se compara com o trabalho desenvolvido.

3.1 *VizAlgo*

Neste artigo, Prabhakar et al. (2022) analisam a eficácia de visualizadores de algoritmos na educação, destacando a plataforma online *VizAlgo*, que permite aos usuários observar o funcionamento interno de algoritmos, como o *backtracking*. A pesquisa enfatiza a importância de uma interface interativa, que visa eliminar a lacuna entre a compreensão teórica e a implementação prática de algoritmos.

Os autores argumentam que a visualização de algoritmos pode melhorar significativamente a qualidade do ensino em Ciência da Computação, conforme evidenciado por questionários que apoiam suas conclusões. A plataforma é projetada para ser acessível, permitindo que os usuários gerem entradas aleatórias e ajustem a velocidade das demonstrações, além de funcionar *offline* e ser leve em termos de espaço.

O artigo também menciona que, apesar dos benefícios, ainda existem desafios na utilização de visualizadores, como a necessidade de implementar representações unificadas e a importância de os alunos praticarem a codificação em uma linguagem de programação escolhida. Os autores concluem que, embora a visualização de algoritmos esteja em estágios iniciais, ela pode ser uma ferramenta poderosa para estudantes curiosos e pode ser expandida para incluir mais algoritmos e estruturas de dados no futuro.

Alguns requisitos deste artigo foram explorados no trabalho atual, tais como o uso navegador *web* para exibir a representação gráfica, contudo a interação com o código-fonte do usuário tem como base formar um diagrama de objetos a partir de requisições *HTTP* ao servidor, enquanto que o *VizAlgo* processa no próprio navegador o código-fonte

e o resultado é obtido para execução no ambiente *desktop*. Além disso, a visualização desenvolvida pelos autores é voltada para solução de tabuleiros do jogo *Sudoku* (Figura 3.1), bem como mostrar o passo a passo de algoritmos de ordenação (Figura 3.2). O recurso de *backtracking* não foi explorado, mas pode fazer parte de trabalhos futuros.

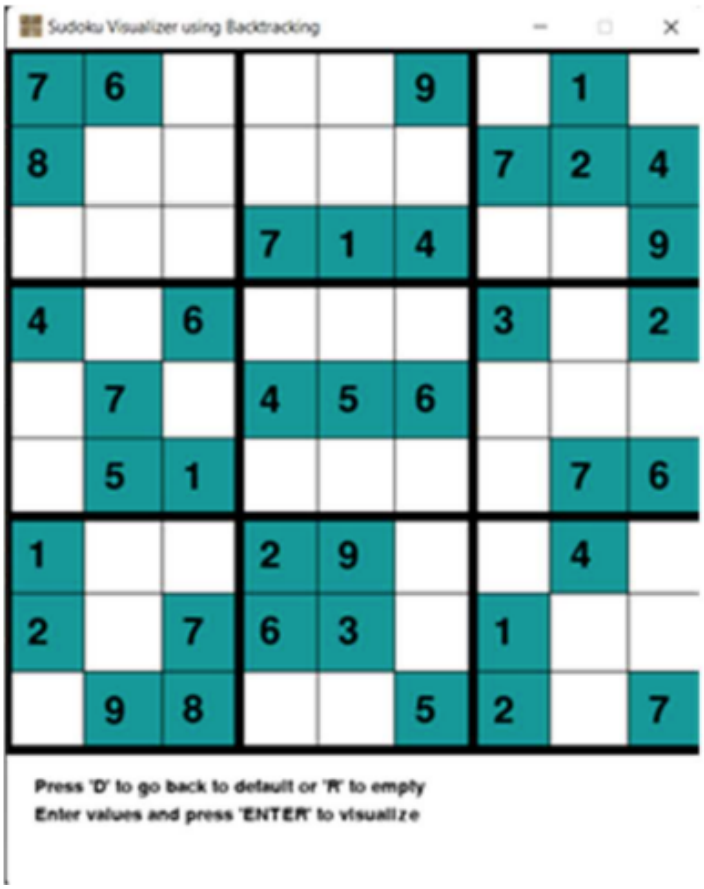

Figura 3.1: *VizAlgo* - Solução do Sudoku (PRABHAKAR et al., 2022)

3.2 *Interactive System for Algorithm and Data Structure Visualization*

O trabalho desenvolvido por Perhác e Simonák (2022) se concentra na concepção, implementação e avaliação de um novo sistema de visualização de algoritmos. O estudo compara brevemente as ferramentas e bibliotecas existentes com base nas visualizações e funcionalidades que oferecem, identificando lacunas que levaram à decisão de desenvolver um novo sistema para visualização de algoritmos e estruturas de dados, destinado ao ensino da disciplina de Estruturas de Dados e Algoritmos.

Figura 3.2: *VizAlgo* - Representação do algoritmo *Bubble Sort* (PRABHAKAR et al., 2022)

O novo sistema foi projetado para ser facilmente utilizável, extensível, disponível e abranger as funcionalidades básicas de sistemas similares, além de incluir outras características úteis. O sistema proposto oferece três tipos de visualizações: visualizações predefinidas para explicar o funcionamento de cada estrutura de dados e algoritmo; visualizações interativas para permitir que o usuário interaja diretamente com a visualização; e exercícios interativos para que os usuários testem seu conhecimento.

Esses três tipos de visualizações abrangem todo o processo de aprendizagem, fornecendo conhecimento teórico e prático, bem como uma maneira de testar o conhecimento adquirido. O sistema foi implementado como uma aplicação web, utilizando a biblioteca *JSAV* para as visualizações. Além disso, o sistema foi avaliado por meio de uma pesquisa com os usuários, e várias melhorias foram incorporadas com base no *feedback* recebido.

A figura 3.3 mostra a execução de um algorimo e sua respectiva representação gráfica. O usuário pode navegar por cada etapa da execução, podendo voltar, avançar ou regular a velocidade de transição do gráfico.

O trabalho destaca a importância das visualizações interativas como uma ferramenta eficaz para melhorar a compreensão e a aprendizagem de algoritmos e estruturas de dados, fornecendo aos alunos uma abordagem prática e envolvente para explorar e compreender conceitos complexos.

No projeto desenvolvido neste trabalho, diferentemente do que foi proposto pe-

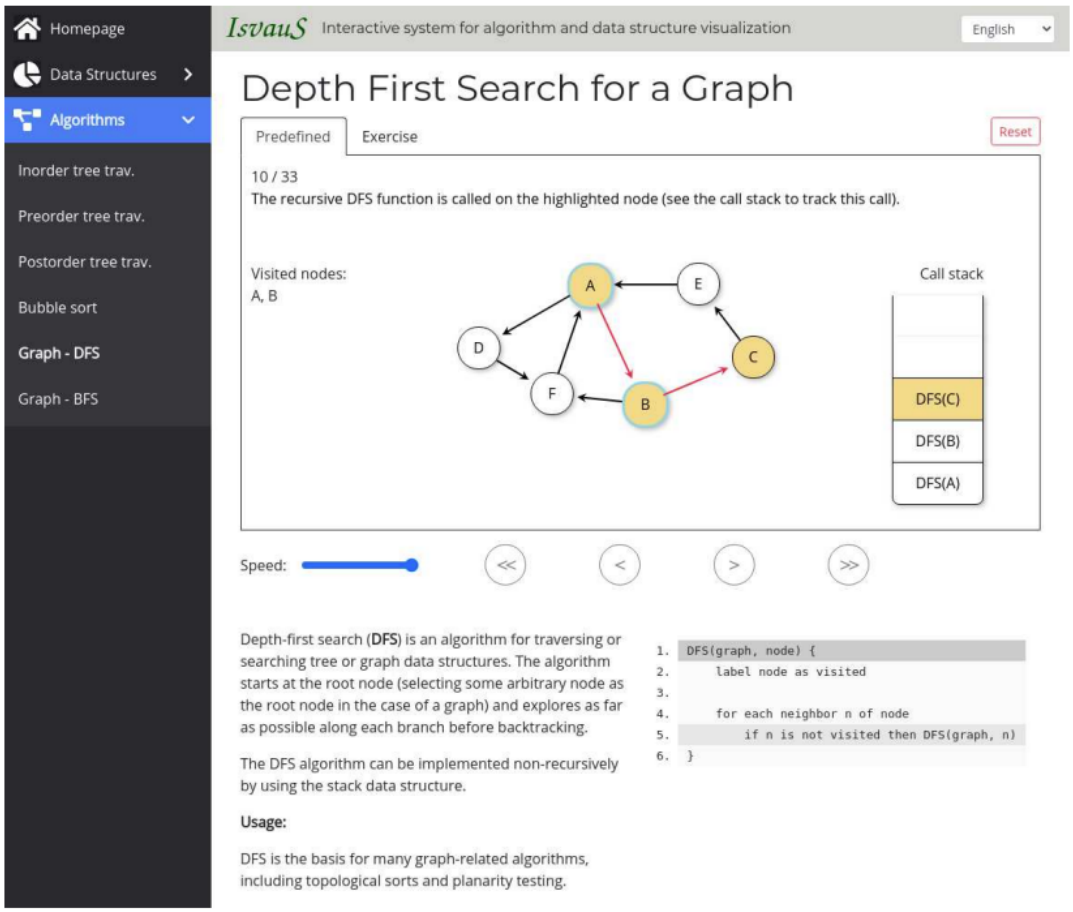

Figura 3.3: Exemplo de representação gráfica (PERHÁC; SIMONÁK, 2022)

los autores, a representação gráfica é formada pelas requisições *HTTP* realizadas pelo código-fonte a ser depurado. Sendo assim, não foi explorado o conceito de visualizações predefinidas ou interatividade com foco em educação, tais como exercícios e textos explicativos. A aplicação pode ser utilizada para representar estruturas de dados, a depender do uso feito pelo desenvolvedor.

3.3 *AlgoAssist*

O artigo apresenta a plataforma *AlgoAssist*, desenvolvida com o objetivo de criar um ambiente de aprendizado interativo que facilite a compreensão de algoritmos e estruturas de dados (*DSA*). A proposta é utilizar visualizações e práticas de codificação para ajudar os alunos a entenderem melhor os conceitos fundamentais da computação. O sistema é estruturado em torno de pré e pós-avaliações, explicações textuais, visua-

lizações linha a linha e um ambiente de codificação, proporcionando uma experiência de aprendizado abrangente (GHANDGE et al., 2021).

O sistema inclui avaliações que medem o conhecimento prévio dos alunos e o aprendizado após a instrução, permitindo que os educadores identifiquem áreas que necessitam de mais atenção. Além disso, são fornecidas breves descrições dos algoritmos antes das visualizações, garantindo que os alunos tenham uma base teórica antes de se aprofundarem na prática. As visualizações de algoritmos são uma parte crucial do sistema, pois ajudam os alunos a entender como cada linha de código manipula os dados, destacando a importância de cada parte do algoritmo e facilitando a compreensão do fluxo de dados.

O *dashboard* da plataforma exibe estatísticas e progresso do aluno, incluindo informações sobre tarefas pendentes e pontuações em avaliações. A interface é projetada para permitir uma navegação fácil entre diferentes seções da aplicação, com opções para atualizar perfis e visualizar notificações. As visualizações interativas incluem botões que permitem aos alunos iniciar, pausar, redefinir e controlar a velocidade da animação, além de um recurso para rastrear a linha de código em execução, tornando o aprendizado mais dinâmico e envolvente. As figuras 3.4 e 3.5 mostram diferentes representações para estruturas de dados e algoritmos, bem como os controles para maior interação do usuário.

Em comparação com outras ferramentas, como o *VizAlgo*, o *AlgoAssist* oferece recursos adicionais, como um portal de atribuições e *quizzes*, um ambiente de codificação e funcionalidades específicas para professores que não estão presentes na ferramenta *VizAlgo*. A conclusão do documento destaca que o sistema visa complementar o aprendizado tradicional, com planos para incluir mais tópicos de Engenharia de Computação, como Aprendizado de Máquina e Inteligência Artificial, além de permitir que os alunos criem suas próprias visualizações de algoritmos.

O objetivo final da plataforma é reduzir o medo dos alunos em relação a *DSA*, tornando o aprendizado mais divertido e envolvente, promovendo uma abordagem ativa e interativa que estimula a curiosidade e o interesse dos estudantes pela computação.

O protótipo desenvolvido pelo trabalho atual não tem por objetivo avaliar alunos por meio de exercícios avaliativos. Apesar de não conter recursos multi-usuário, estes po-

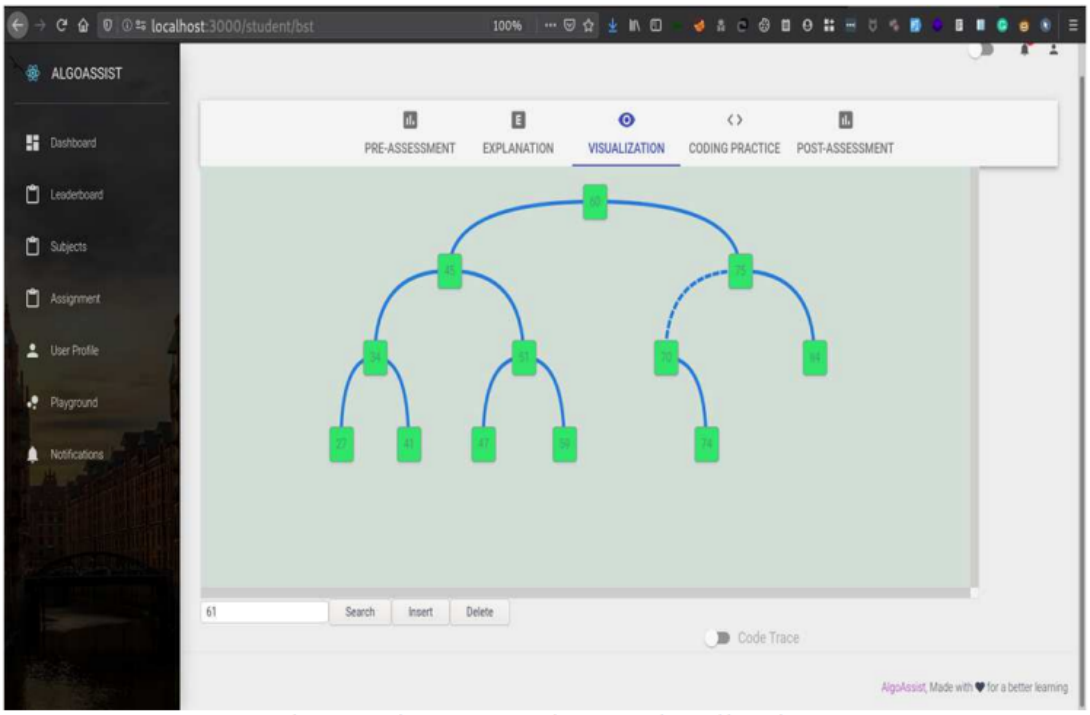

Figura 3.4: *AlgoAssist* - Árvore de Busca Binária (GHANDGE et al., 2021)

dem vir a fazer parte de trabalhos futuros, visando disponibilizar remotamente a aplicação para múltiplos desenvolvedores simultâneos, cada um em seu respectivo ambiente de depuração.

3.4 *AlgoViz*

O projeto *AlgoViz* visa facilitar a compreensão de algoritmos através de visualizações gráficas, utilizando as bibliotecas *Pygame* e *Tkinter* (GUPTA; VYAWAHARE, 2023). O sistema abrange três tipos principais de algoritmos:

1. Algoritmos de Busca: Permite ao usuário inserir um *array* e um elemento a ser buscado, demonstrando como diferentes algoritmos localizam o elemento;

2. Algoritmos de Ordenação: Representa os elementos em um gráfico de barras, onde a execução do algoritmo é visualizada pela alteração do comprimento das barras, mostrando a ordenação em tempo real (Figura 3.6).

3. Algoritmos de Caminho: O usuário pode definir um nó inicial e um nó final em uma grade, além de criar barreiras. O algoritmo explora a grade para encontrar o

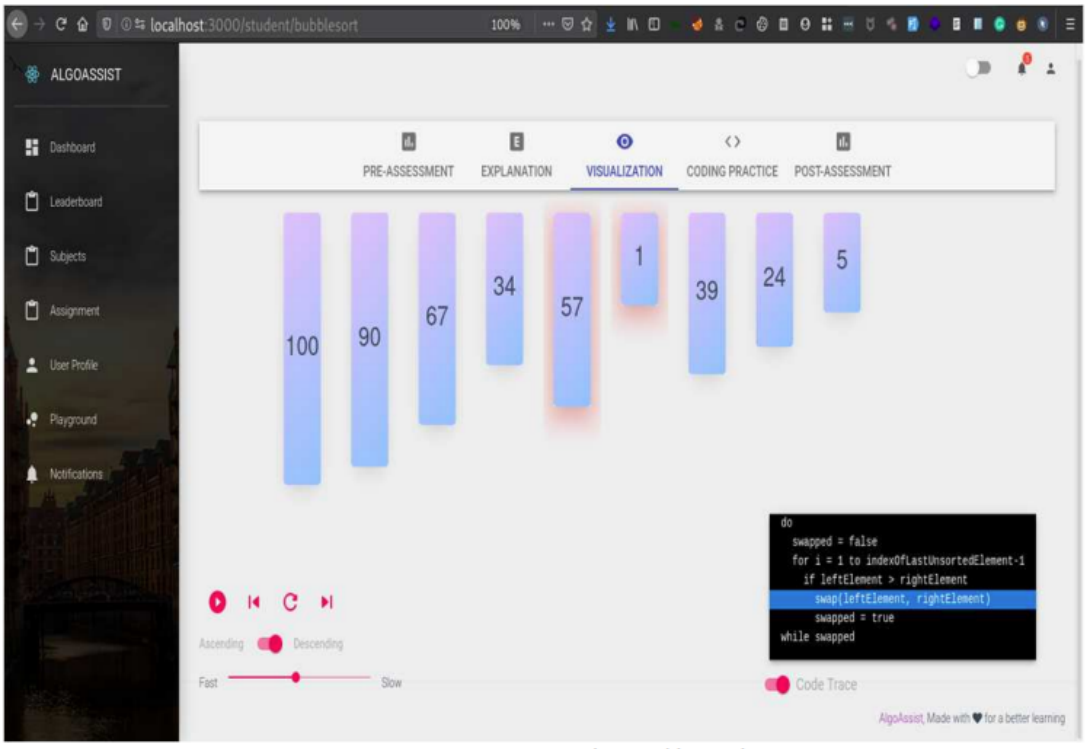

Figura 3.5: *AlgoAssist - Bubble Sort* (GHANDGE et al., 2021)

caminho mais curto, com cada bloco representando um nó.

O sistema é dividido em dois módulos: o módulo principal gerencia a execução e visualização dos algoritmos, enquanto o módulo de plugins contém a lógica dos algoritmos. O projeto conclui que, apesar de existirem boas visualizações, há uma necessidade contínua de melhorias e mais representações visuais para tópicos menos abordados. Sugestões para futuras melhorias incluem a adição de mais algoritmos, controles de execução e a possibilidade de implementação como um aplicativo móvel.

A aplicação desenvolvida no trabalho atual é voltada para o ambiente *web*, o que o torna diferente do *AlgoViz*, que foi criado para o ambiente *desktop*. Os algoritmos trabalhados podem ser testados no projeto atual, desde que as devidas conexões ao servidor sejam feitas para representar graficamente os elementos da estrutura de dados e atribuir cores e alterações de valor, de acordo com o processamento feito pelo código-fonte.

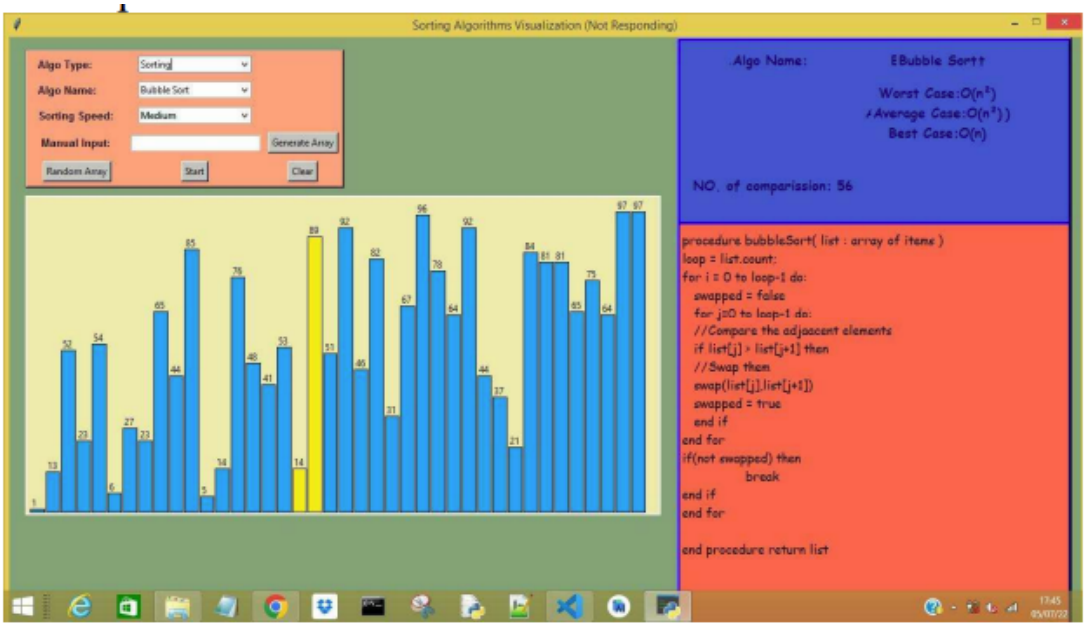

Figura 3.6: *AlgoViz - Bubble Sort* (GUPTA; VYAWAHARE, 2023)

3.5 *AlgoRhythm*

AlgoRhythm é uma ferramenta de visualização projetada para aprimorar as metodologias de ensino de algoritmos de ordenação e de busca de caminho. O foco da ferramenta é visualizar diversos algoritmos em um quadro, proporcionando uma experiência de aprendizado mais interativa e envolvente para os alunos (TRIVEDI et al., 2023).

Os principais recursos observados no *AlgoRhythm* são:

- Visualização Interativa: A ferramenta permite que os usuários visualizem o funcionamento de diferentes algoritmos de ordenação (como *Bubble Sort* (Figura 3.7), *Quick Sort*, *Merge Sort*) e algoritmos de busca de caminho (como *Dijkstra* e A*). Esse aspecto interativo ajuda os alunos a entender melhor a mecânica desses algoritmos;

- Aprimoramento Educacional: Ao visualizar algoritmos, o *AlgoRhythm* visa melhorar a compreensão e a retenção dos conceitos algorítmicos pelos alunos, levando a resultados de aprendizado potencialmente mais eficazes em comparação com os métodos tradicionais de ensino;

- Arquitetura: A arquitetura do *AlgoRhythm* é projetada para suportar diversos algoritmos, tornando-o uma ferramenta versátil para educadores e aprendizes. Ela

transcende as metodologias existentes ao integrar visualização com a educação algorítmica, promovendo o aprendizado ativo.

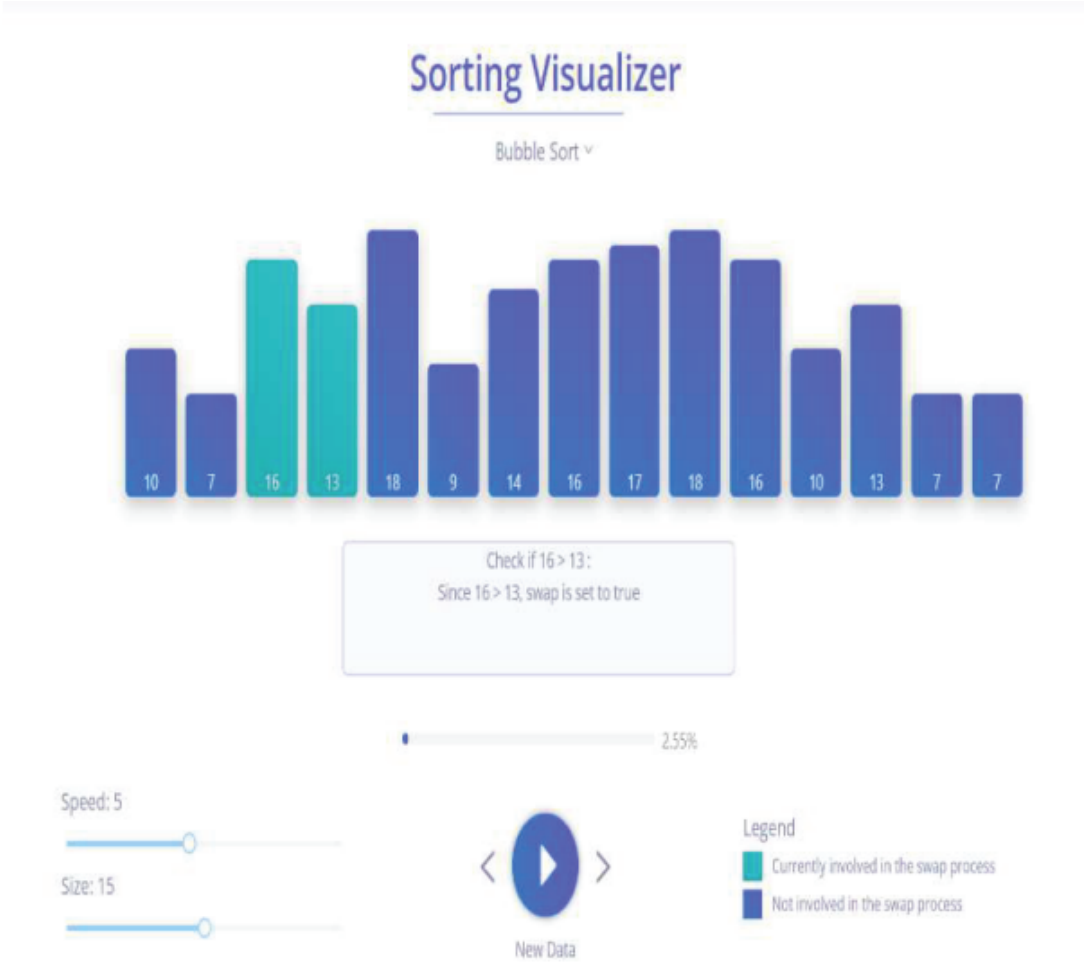

Figura 3.7: *AlgoRhythm* - Algoritmo de Ordenação (TRIVEDI et al., 2023)

Os autores destacam a eficácia da ferramenta em melhorar as metodologias de ensino de algoritmos e seu impacto potencial nos resultados de aprendizado dos alunos.

Em resumo, o *AlgoRhythm* representa um avanço significativo nas ferramentas educacionais disponíveis para o ensino de algoritmos, aproveitando a visualização para fomentar uma compreensão mais profunda de conceitos complexos em Ciência da Computação.

O trabalho desenvolvido permite a visualização de tais algoritmos, a partir de requisições *HTTP* feitas a partir do código-fonte do desenvolvedor. Assim, este pode gerar os nós da árvore ou grafo e demarcar por meio de cores o estado de cada elemento percorrido.

3.6 *TreEducation*

TreEducation é uma plataforma educacional projetada para melhorar a alfabetização em visualização de dados, especificamente utilizando *treemaps*. A plataforma permite que os alunos criem seus próprios *treemaps*, recebendo *feedback* imediato sobre suas escolhas, conforme a figura 3.8 (FUCHS et al., 2024).

As principais características da ferramenta incluem:

- Algoritmos de Layout: A plataforma oferece múltiplos algoritmos de *layout*, como *Slice-and-Dice* e *Squarify*, permitindo que os alunos comparem e entendam as diferenças entre eles;
- *Gamificação*: Elementos de *gamificação* foram incorporados, como *quizzes* e suporte didático, para aumentar a motivação e o engajamento dos alunos durante o aprendizado;
- Avaliação: O *TreEducation* foi testado em ambientes de sala de aula e controlados, utilizando testes de alfabetização em *treemaps* já estabelecidos.

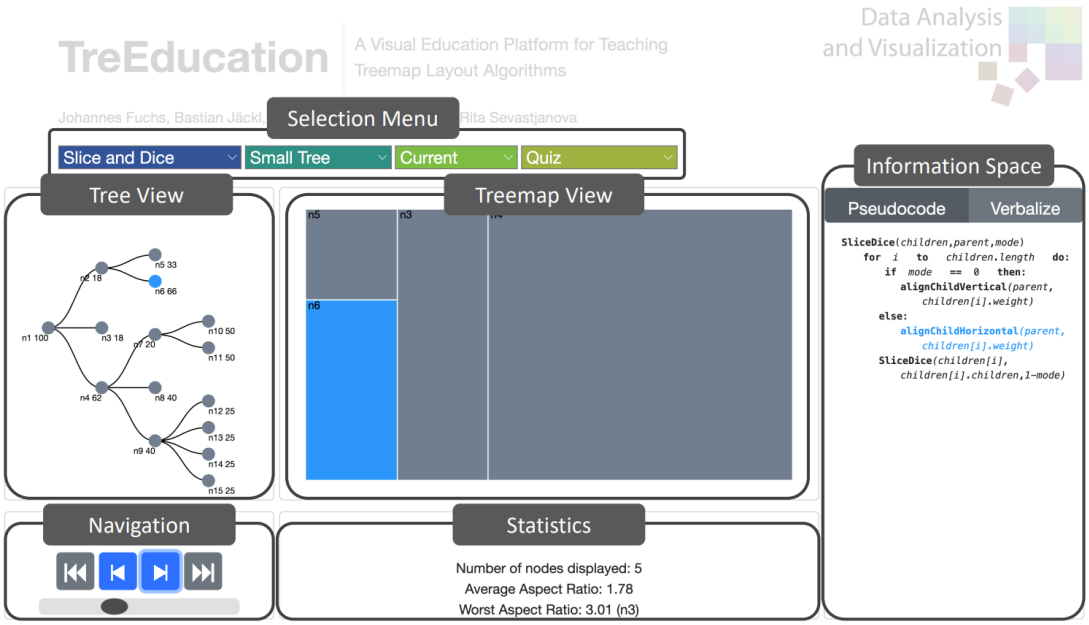

Figura 3.8: *TreEducation - Treemaps* (FUCHS et al., 2024)

Os resultados obtidos indicam um ganho significativo de conhecimento entre os alunos que utilizaram o *TreEducation*, demonstrando a eficácia da plataforma em melhorar

a compreensão de visualizações de dados. A pesquisa sugere que a abordagem interativa e *gamificada* é mais eficaz do que métodos tradicionais de ensino.

Em relação ao trabalho aqui apresentado, a aplicação se apoia na eficácia da abordagem interativa, ao permitir que o desenvolvedor possa visualizar dados em tempo real. Contudo, não foram trabalhados os conceito de *gamificação* e avaliação de alunos pela plataforma, presentes no resultado obtido pelos autores.

4 Metodologia

4.1 Requisitos

Os seguintes requisitos foram levados em conta durante a construção da aplicação.

4.1.1 Requisitos Funcionais

- Gerenciar (criar, editar e excluir) entidades e relacionamentos, ou seja, nós e arestas, por meio de uma *API REST*. Com isso, programadores de diversas linguagens poderão se comunicar com o servidor por meio de requisições *HTTP*;
- Visualizar graficamente a relação destes elementos em um navegador *web*, o que torna a aplicação mais acessível para os usuários;
- Ajuste de câmera para exibir todos os elementos, tanto automático quanto disponível em um botão para o usuário;
- Atualizar representação gráfica em tempo real por meio de um *WebSocket*;
- Disponibilizar classe ou biblioteca que abstraia as requisições *REST* feitas pela aplicação a ser analisada ao servidor;
- Dados da aplicação devem ser persistidos em um SGBD.

4.1.2 Requisitos Não-Funcionais

- Capacidade de adicionar novas funcionalidades ou modificar as existentes de forma eficiente e com impacto mínimo em outras partes da aplicação.
- Utilizar bibliotecas e ferramentas de representação gráfica com potencial para aprimoramentos futuros, a partir dos diversos recursos visuais disponíveis.

4.2 Design

As entidades que compõem o modelo do grafo são Nó e Aresta. A primeira deve conter rótulo a ser exibido (*label*) e cor (*color*), que por padrão assume o valor "*white*". Já a segunda deve conter referências ao nó de origem (*source*) e destino (*target*), bem como um rótulo (*label*, que por padrão é uma sequência de caracteres vazia. Ambos devem conter chave primária (*id*).

4.3 Arquitetura

A aplicação foi construída com base na arquitetura cliente-servidor, conforme a Figura 4.1.

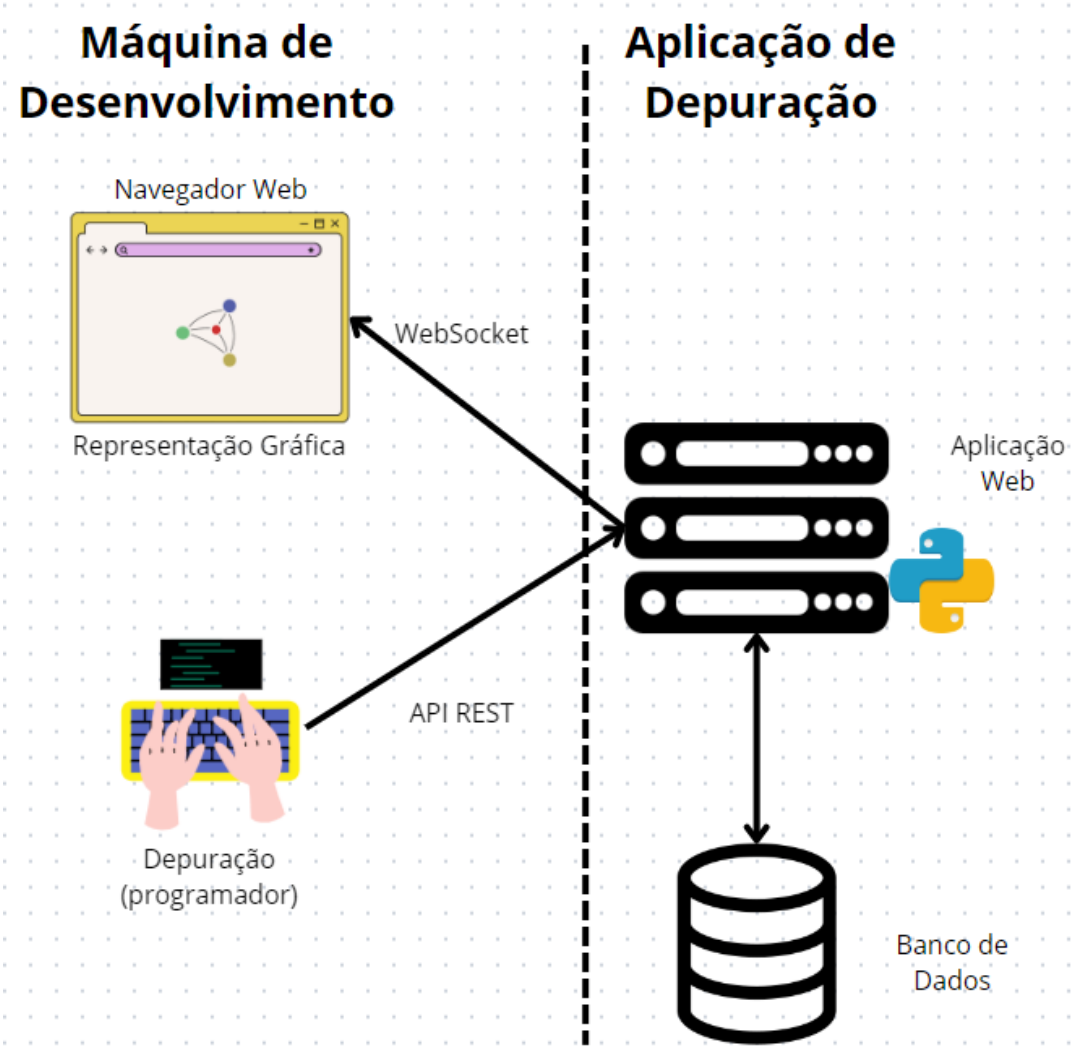

Figura 4.1: Arquitetura da aplicação

O programador adiciona ao seu código-fonte as requisições ao servidor remoto, por meio da biblioteca de conexão. As requisições são responsáveis por criar nós e arestas à representação gráfica.

Os elementos gráficos podem ser utilizados para representar qualquer dado, objeto ou estrutura de dados, de acordo com a necessidade do usuário. Ou seja, não apenas utilizar o modelo para representar grafos, mas quaisquer estruturas de dados, bem como diagramas de objetos. No primeiro caso, os nós representam os dados da estrutura e as arestas o encadeamento entre eles; no último caso, as arestas poderiam representar relacionamentos entre os objetos.

Em seguida, o desenvolvedor acessa a representação gráfica por um navegador *web*. A aplicação *Web* fornece uma página estática, que permite visualizar do modelo. O servidor também processa as informações recebidas à cada requisição feita à *API REST*, armazenando os dados em um SGBD e em seguida atualiza em tempo real o navegador por meio de um *WebSocket*.

REST é uma abordagem simples e flexível para a comunicação entre sistemas usando o protocolo *HTTP*. Ele permite que os clientes acessem e manipulem recursos no servidor de forma eficiente por meio de uma *API*. Logo, o sistema permite testar códigos desenvolvidos em qualquer linguagem de programação que realize requisições a partir protocolo *HTTP*.

4.4 Linguagens e Tecnologias Utilizadas

A infraestrutura utilizada para o desenvolvimento do repositório compreende:

4.4.1 *Backend*

O *Python* será a linguagem principal, aliado ao *framework Django*. A escolha considera:

- Ecossistema e Suporte: A robusta comunidade e a vasta gama de recursos tornam o *Python* uma escolha sólida.
- Facilidade de Integração com o Banco de Dados: O ambiente possui excelentes

bibliotecas para integração com bancos de dados relacionais, facilitando a interação com o mesmo.

- Facilidade de Manutenção e Escalabilidade: A robustez do *Django* proporcionam código legível, testável e escalável.

- Integração com Tecnologias *Frontend*: O *framework* oferece, por padrão, um mecanismo baseado em *templates* para fornecer páginas *web*. Além disso, possui diversas opções para integração com *frameworks* e bibliotecas *JavaScript* populares.

Além disso, o *Django REST Framework* — presente no *framework* escolhido — possui um conjunto poderoso e flexível de ferramentas para construir *APIs REST*. Com isso, permite gerenciar os dados do modelo da aplicação via requisições *HTTP* via *web*.

4.4.2 *Frontend*

Para criar a interface de usuário, foram utilizados *HTML*, *CSS* e *JavaScript*. O código-fonte foi criado a partir de *templates*, que são processadas e entregues pelo *framework Django*.

A biblioteca *Three.js* foi empregada para representar graficamente os elementos. Para representar o grafo, os seguintes módulos foram utilizados:

- *TextGeometry*: permite criar elementos gráficos em forma de texto. Neste projeto, é utilizado para representar o rótulo do nó representação 3D;

- *FontLoader*: necessário para importar estilos de fonte no modelo;

- *lil-gui*: permite criar interface gráfica com opções que facilitem a interação com a representação;

- *3d-force-graph*: biblioteca externa que permite criar grafos tridimensionais. Possui diversos recursos gráficos e permite personalizar a representação de nós e arestas (exemplo na Figura 4.2);

- *force-graph*: biblioteca externa, equivalente a anterior para contextos bidimensionais (exemplo na Figura 4.3);

- *CSS2DRenderer*: biblioteca externa, útil para representar elementos bidimensionais em um contexto 3D;

- *three-spritetext*: biblioteca externa, que permite criar elementos textuais na representação gráfica.

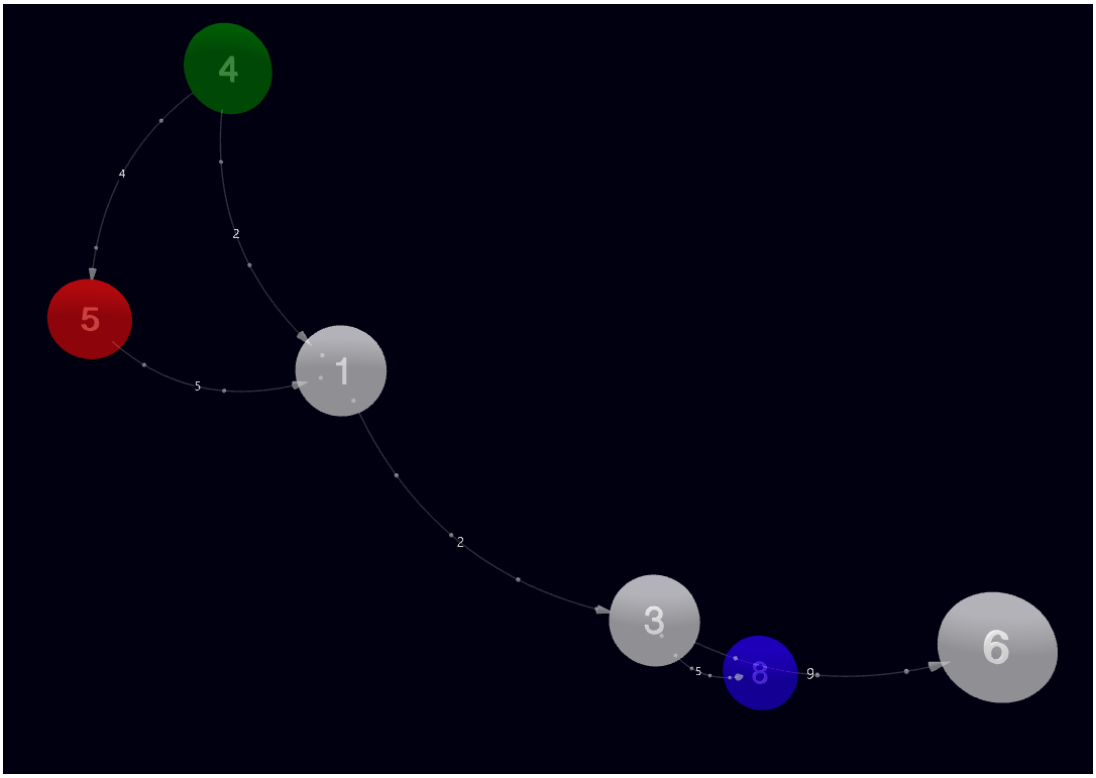

Figura 4.2: Representação em 3D

4.4.3 Banco de Dados

A modelagem do banco de dados segue o modelo relacional. O SQLite foi utilizado no ambiente de desenvolvimento, com a possibilidade de utilizar, futuramente, outros SGBDs mais robustos para os ambientes de homologação e produção.

O uso de Mapeamento Objeto-Relacional (*ORM*) facilita o gerenciamento e persistência de dados. Assim, é possível recuperar tuplas de uma tabela do modelo como objetos na aplicação. Além disso, consultas e alterações de registro passam a ser realizadas a partir dos métodos disponíveis no *ORM*, que funcionam independente de qual SGBD estiver em uso.

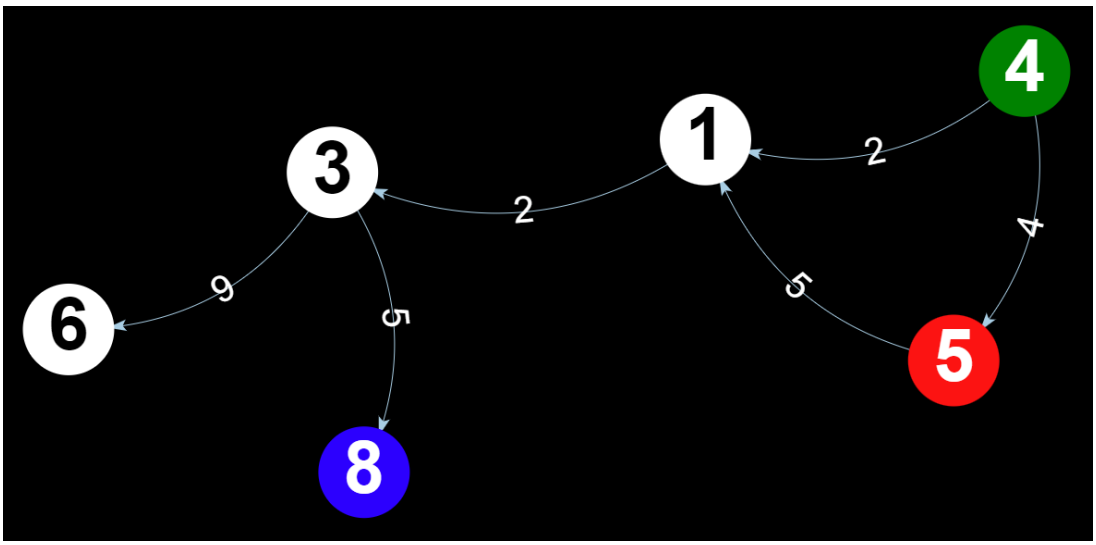

Figura 4.3: Representação bidimensional do grafo

O banco de dados persistirá as informações de cada nó e aresta a cada requisição feita pelo usuário (inserção, atualização e exclusão).

4.4.4 Classe/biblioteca de conexão

Foi utilizada a linguagem de programação *Java* para criar um exemplo de classe de conexão, que possa ser utilizado em códigos desta linguagem. Este processo pode ser repetido para qualquer outra linguagem de programação.

As conexões feitas ao servidor a partir da biblioteca de conexão devem seguir a especificação da *API REST* desenvolvida, disponível em ⟨https://github.com/ufjf-joaopauloaraujo/graph_api/blob/main/examples/openapi.json⟩.

4.4.5 Servidor Web

Durante o desenvolvimento foi adotado o WSGI, embutido por padrão no framework Django, para executar a aplicação. Outras opções podem ser utilizadas em produção, tais como *Apache*, *Nginx* ou *Gunicorn*.

4.4.6 Ferramentas de Desenvolvimento

Será utilizada como ambiente de desenvolvimento integrado (*IDE*) o *Visual Studio Code*, facilitando o desenvolvimento, depuração e controle de versão com o *Git*.

O código-fonte encontra-se disponível na plataforma *GitHub* no seguinte repositório: ⟨https://github.com/ufjf-joaopauloaraujo/graph_api.git⟩.

4.5 Implementação

O gerenciamento de nós e arestas está disponível por meio de requisições *REST*, disponibilizadas utilizando *Django REST Framework* (*DRF*). Como destaque, foram viabilizados os métodos de criar, editar e excluir nós e arestas, bem como a exclusão de todos os elementos para reiniciar a representação de dados. A definição destes métodos se encontra no arquivo ⟨https://github.com/ufjf-joaopauloaraujo/graph_api/blob/main/graph_api/views.py⟩.

A integridade dos dados é garantida ao se excluir em cascata todas as arestas cuja origem ou destino são excluídas. As definições do modelo foram definidas no arquivo ⟨https://github.com/ufjf-joaopauloaraujo/graph_api/blob/main/graph_api/models.py⟩.

A classe de conexão, escrita em *Java*, cobre todas estas requisições. Nos casos em que há valores opcionais, há sobrecarga de métodos, permitindo a omissão de valores opcionais. Esta se encontra disponível em ⟨https://github.com/ufjf-joaopauloaraujo/graph_api/blob/main/examples/examples/GraphAPI.java⟩.

A representação gráfica foi construída utilizando a biblioteca *Three.js*, onde foi possível construir representações gráficas em 2D e 3D. Todas as configurações e definições do modelo estão disponíveis em ⟨https://github.com/ufjf-joaopauloaraujo/graph_api/blob/main/graph_api/static/js/main.js⟩.

Os nós são representados como círculos (ou esferas), podendo escolher a cor e o texto; 12 cores foram enumeradas na classe de conexão, tanto para auxiliar a escolha do usuário quanto na definição da cor da fonte a ser utilizada na representação bidimensional. Já as arestas são representadas como setas, com a descrição (opcional) ao centro.

Na representação tridimensional foram adicionados recursos visuais opcionais, disponíveis na biblioteca *3d-force-graph*, tais como:

- Partículas nas arestas, que facilitam a visualização do sentido que apontam;
- Modo de foco, onde o usuário pode clicar em um nó e a câmera se posicionará a

partir da perspectiva do mesmo;

- Modo de texto, onde os nós serão representados apenas pelo texto que os descreve;
- Ajuste automático de câmera, onde esta ajusta o *zoom* de forma que exiba todos os elementos do grafo; este ajuste é realizado à cada requisição, podendo ser feito pelo usuário com a opção "*Restart Camera*".

A transição entre a representação 3D e 2D pode ser feita a partir da opção *3dMode*. Os recursos de visualização de partículas e ajuste automático de câmera também estão disponíveis na versão bidimensional.

Uma interface gráfica permite escolher entre as representações visuais e habilitar os recursos visuais. Esta se encontra definida em ⟨https://github.com/ufjf-joaopauloaraujo/graph_api/blob/main/graph_api/templates/graph_ui.html⟩.

A comunicação entre o *backend* e o navegador *web*, a cada requisição feita, é realizado por *WebSocket*, atualizando-se imediatamente a representação gráfica. O servidor *WebSocket Daphne* é utilizado para facilitar a comunicação entre a aplicação e o *browser*. Além disso, a persistência das mensagens a serem enviadas por esta conexão é feita em memória, sem o uso de outra aplicação externa, como o *Redis*, no intuito de facilitar o desenvolvimento inicial.

A configuração do *WebSocket* no *backend* se encontra em ⟨https://github.com/ufjf-joaopauloaraujo/graph_api/blob/main/graph_api/consumers.py⟩.

5 Resultados

A versão inicial da aplicação foi desenvolvida com sucesso, visando atender aos requisitos listados. O código-fonte encontra-se disponível na plataforma *GitHub* no seguinte repositório: ⟨https://github.com/ufjf-joaopauloaraujo/graph_api.git⟩.

5.1 Casos de Uso

Para ilustrar o uso da aplicação, considere os cenários a seguir. O passo-a-passo foi obtido a partir do uso de *breakpoints* durante a depuração do código.

É necessário importar a classe de conexão, conforme a Figura 5.1. Ao referenciar a instância desta, podemos realizar requisições ao servidor remoto e, em seguida, observar os resultados no navegador *web*. Alguns exemplos se encontram na Figura 5.2.

```
import examples.GraphAPI;
import examples.GraphAPI.VertexColor;
```

Figura 5.1: Importação da classe de conexão em Java

```
GraphAPI.getInstance().resetModel(); // Reiniciar modelo - exclui todos os nós e arestas
GraphAPI.getInstance().createVertex(name:"1"); // Cria nó
```

Figura 5.2: Requisições ao servidor a partir da instância da classe de conexão

Para que não seja necessário depurar o código-fonte, foi adicionado à classe de conexão o atributo *delay*, que corresponde ao tempo, em milissegundos, que o método terá que esperar antes de realizar a requisição ao servidor. Para alterar o valor padrão (*0 ms*), basta utilizar o método *setDelay()*, conforme a Figura 5.3.

```
GraphAPI.getInstance().setDelay(delay:3000); // tempo em milissegundos
```

Figura 5.3: Método *setDelay()*

Também foi acrescentado o atributo *disabled*, que determina se as requisições feitas pela classe de conexão devem ser processadas. Ao utilizar o método *disable()* (Figura 5.4), todas as requisições ao servidor serão ignoradas. Esse comando permite que as chamadas ao servidor remoto possam ser mantidas no código-fonte quando as requisições não forem necessárias.

```
GraphAPI.getInstance().disable();
```

Figura 5.4: Método *disable()*

5.1.1 *QuickSort*

O *QuickSort* é um algoritmo de ordenação eficiente, onde se utiliza um método de partição. Nele, um elemento é escolhido como pivô e a lista é reorganizada de tal maneira que os elementos menores que o pivô ficam à esquerda, e os maiores à direita. Em seguida, aplica recursivamente o mesmo processo nas sub-listas resultantes (DROZDEK, 2012).

A linha do tempo apresentada nas figuras 5.5, 5.6, 5.7 e 5.8 mostra a execução deste algoritmo sendo representado graficamente no *browser*. O algoritmo utilizado encontra-se disponível no repositório Git: ⟨https://github.com/ufjf-joaopauloaraujo/graph_api/blob/main/examples/QuickSort.java⟩.

Para que seja possível referenciar nós, é necessário que o seu identificador numérico seja armazenado em uma outra estrutura de dados, conforme a segunda etapa da figura 5.5. Esta referência é utilizada na criação de arestas e na alteração das propriedades visuais do elemento. O gerenciamento de arestas, dentro do mesmo raciocínio, deve ser feito utilizando-se o identificador retornado.

Ao criar cada nó, foi utilizada a cor padrão (branca). A cada alteração, cores arbitrárias foram utilizadas para discernir a etapa de execução; nesse caso, a escolha do pivô ou as trocas de valor entre índices da lista.

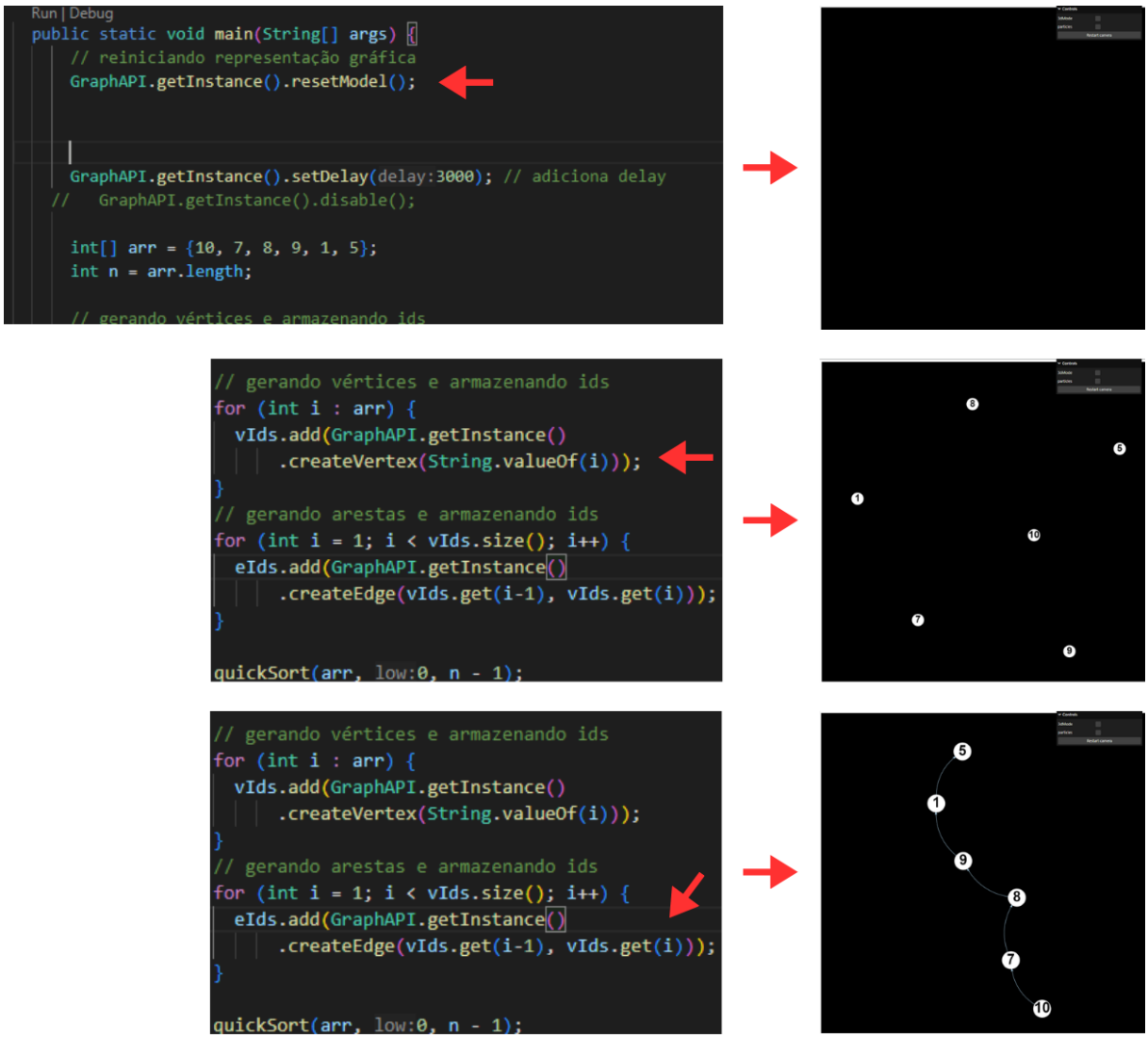

Figura 5.5: Quicksort - Representação inicial do *array*

5.1.2 Busca em Profundidade

A Busca em Profundidade (*Depth-First Traversal - DFT*) é um dos métodos mais usados para explorar estruturas de dados em forma de árvore. O principal objetivo da *DFT* é visitar todos os nós (ou vértices) de maneira que se desça o mais fundo possível em um ramo antes de voltar (retroceder) para explorar outros ramos (DROZDEK, 2012).

As figuras 5.9, 5.10, 5.11 e 5.12 mostram o passo-a-passo de processamento deste algoritmo ao se utilizar da ferramenta gráfica. O código encontra-se disponível em: ⟨https://github.com/ufjf-joaopauloaraujo/graph_api/blob/main/examples/Grafo.java⟩.

Sendo a raiz da árvore o nó "0", a cada execução do método recursivo *DSFUtil*, temos a alteração de cor para demarcar o último elemento percorrido.

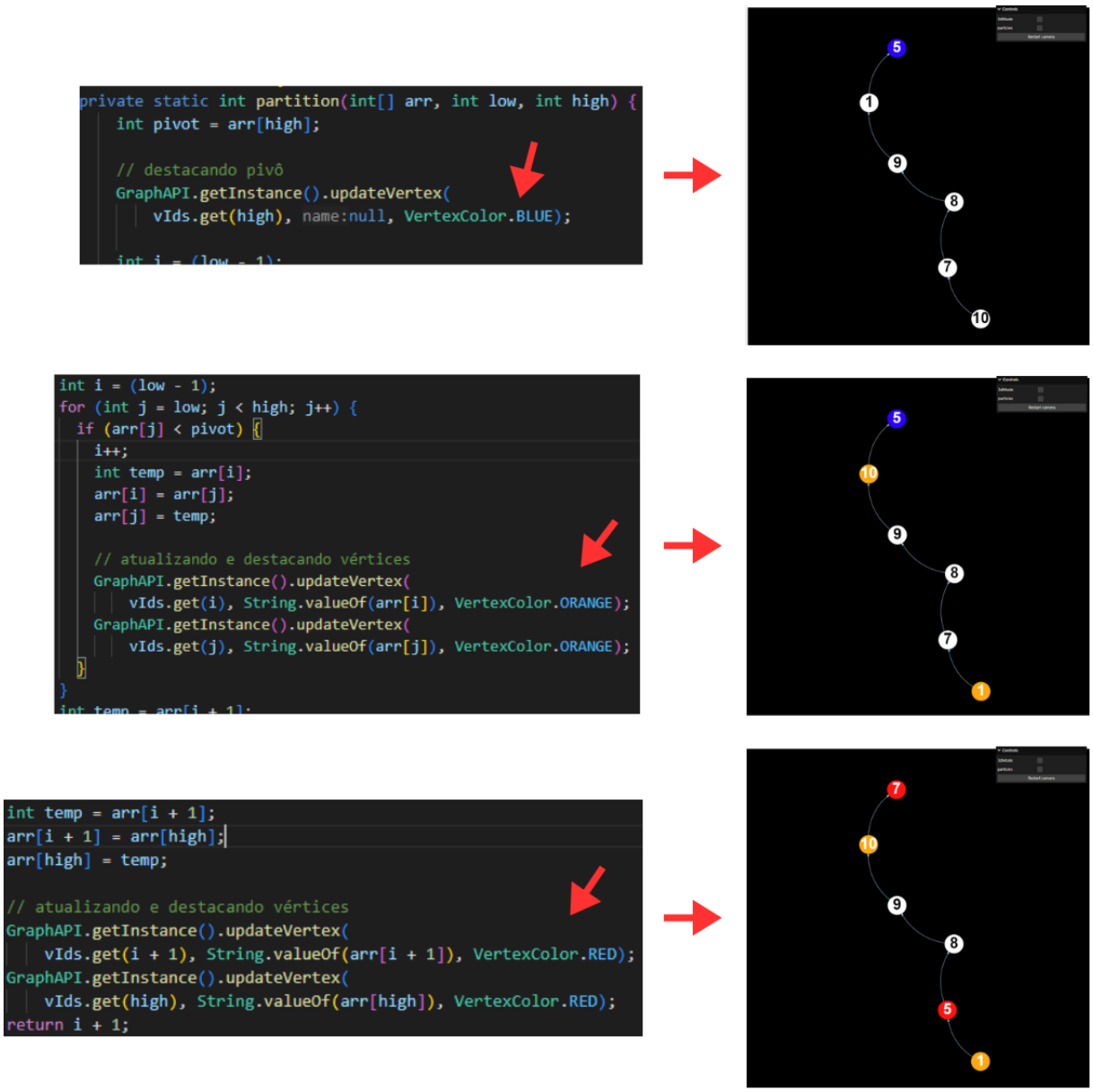

Figura 5.6: Quicksort - Escolha do pivô e primeiras ordenações

A criação das arestas é feita individualmente no método *main*, referenciando cada par origem-destino individualmente, conforme a Figura 5.9.

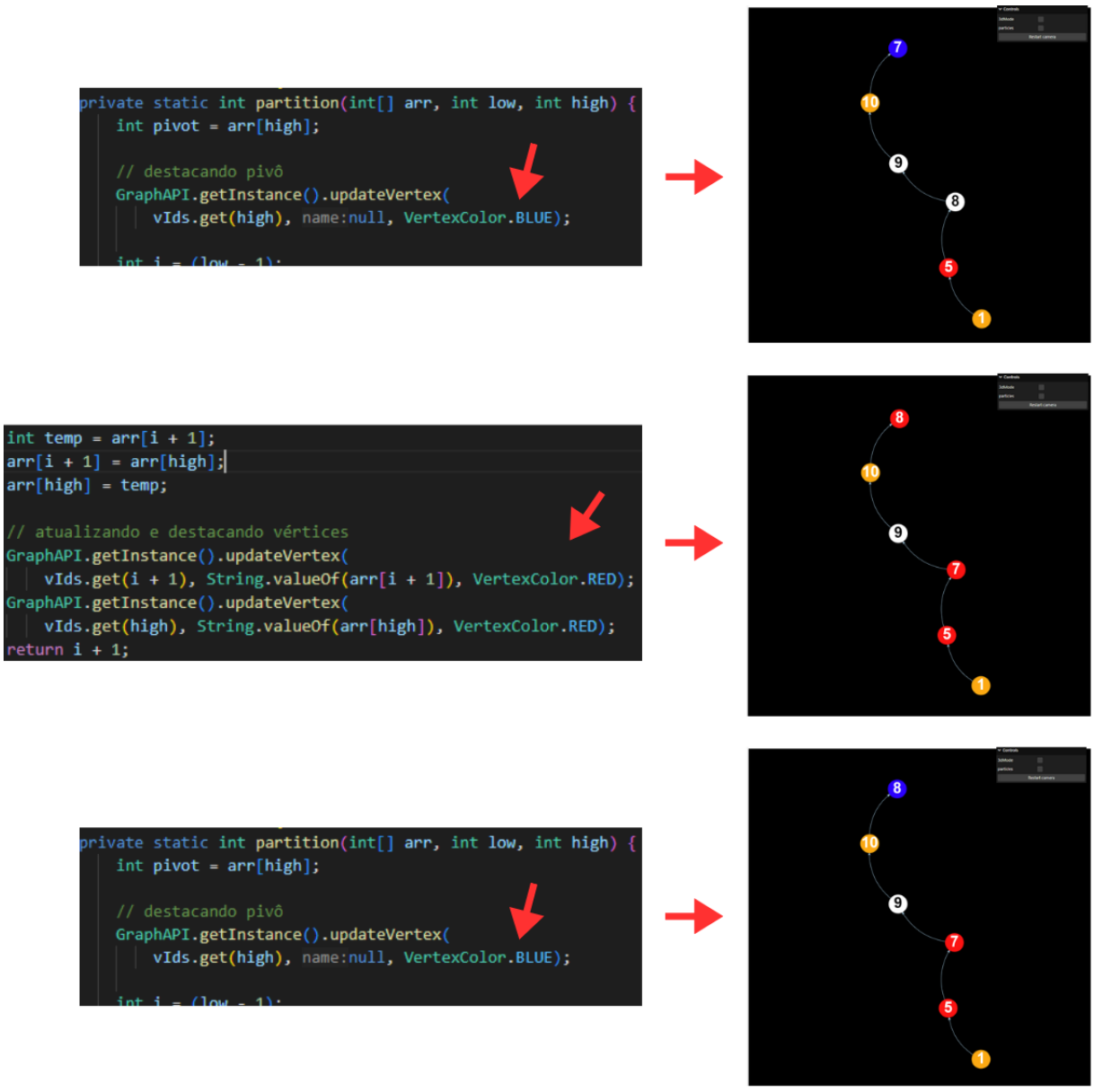

Figura 5.7: Quicksort - continuação

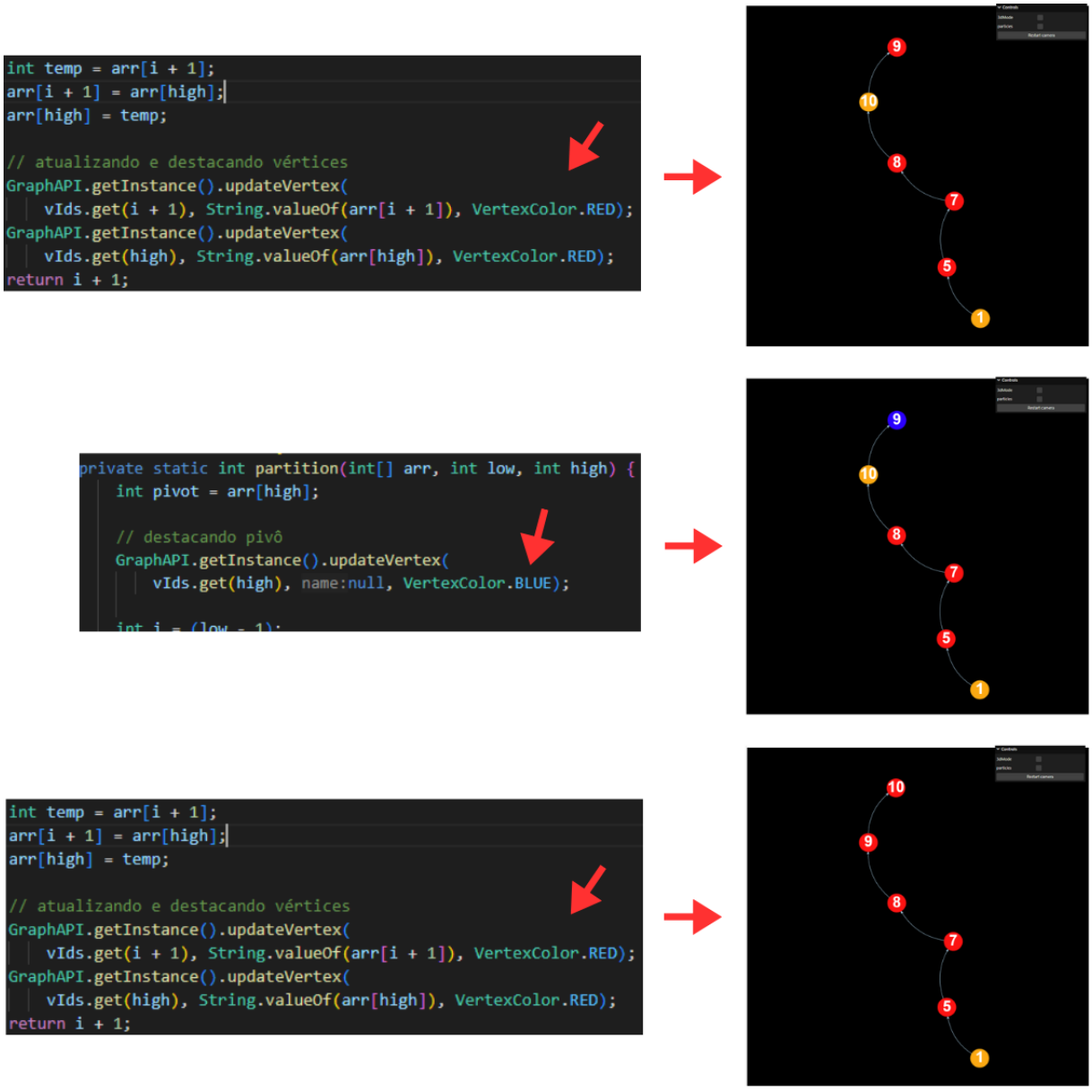

Figura 5.8: Quicksort - Dados ordenados pelo algoritmo

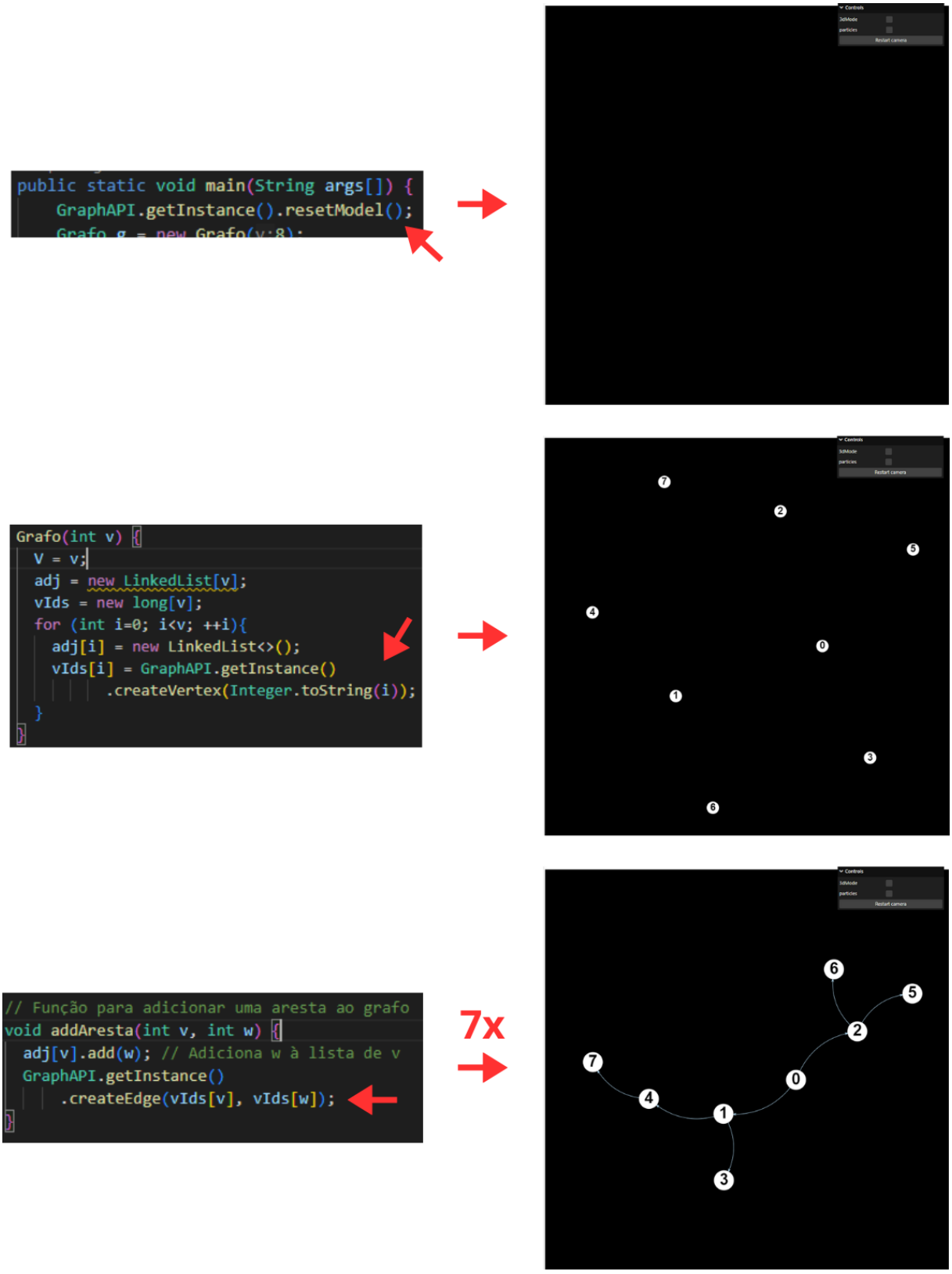

Figura 5.9: DFT - Representação inicial da árvore binária

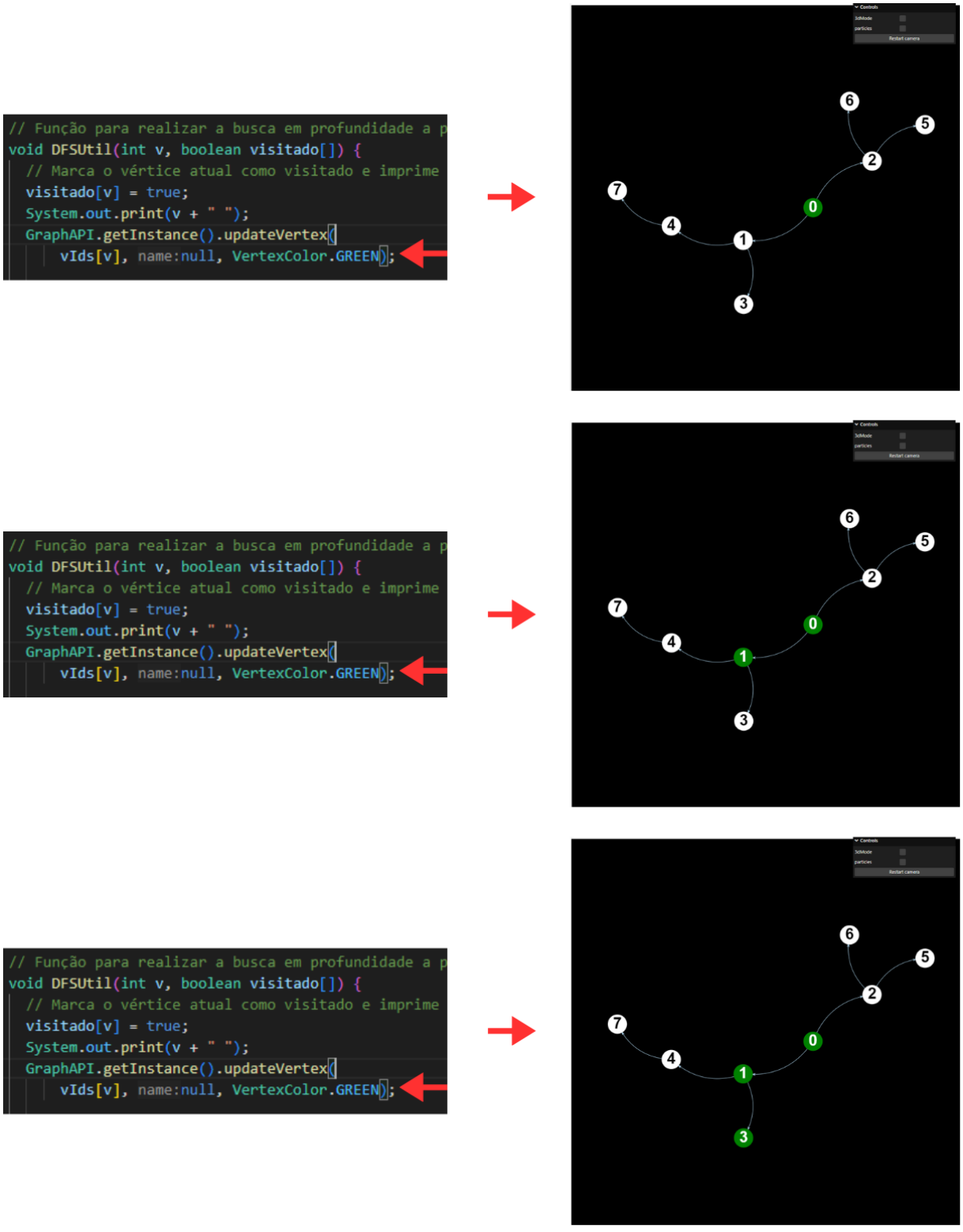

Figura 5.10: DFT - Percorrendo árvore a partir da raiz

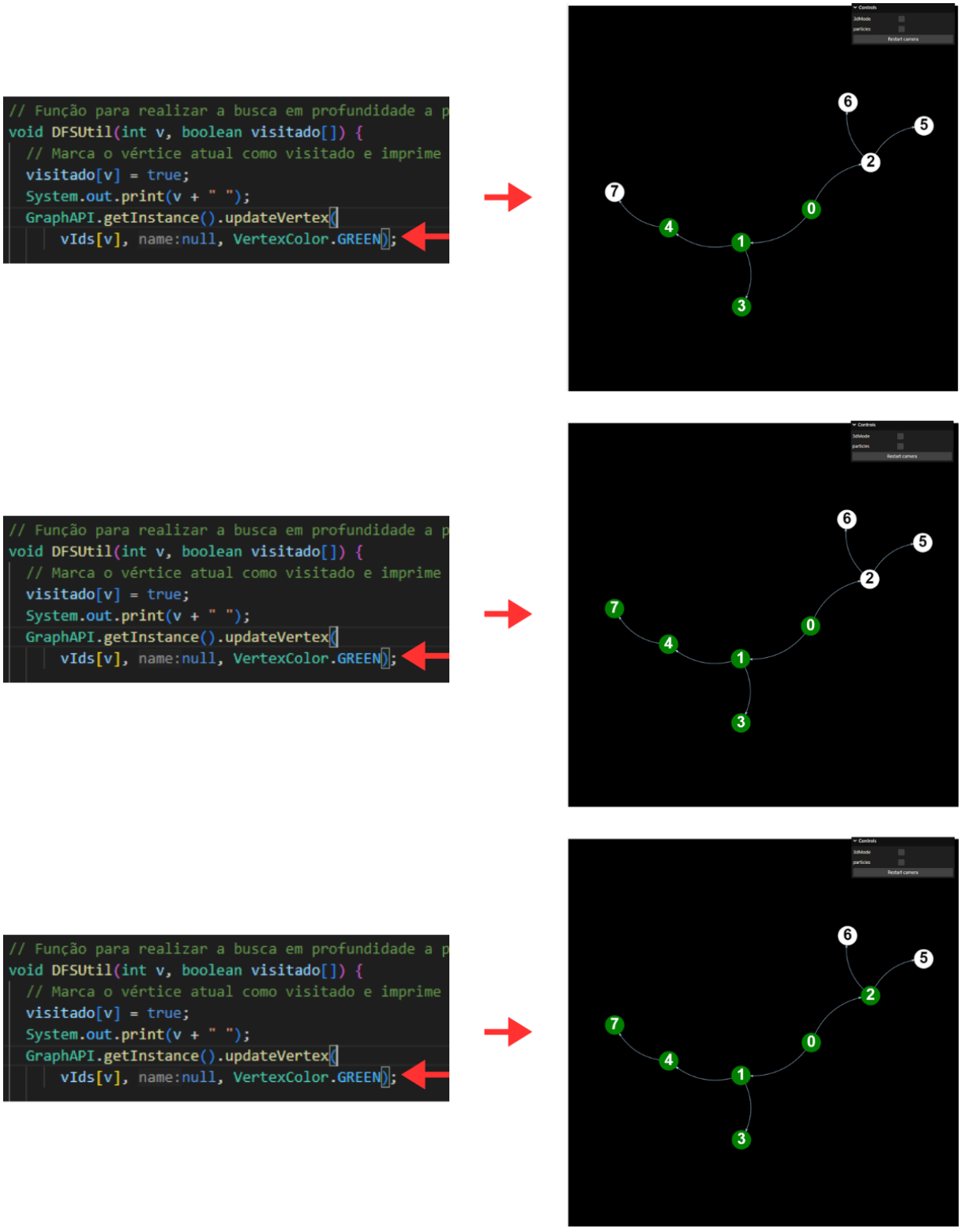

Figura 5.11: DFT - Continuação do algoritmo

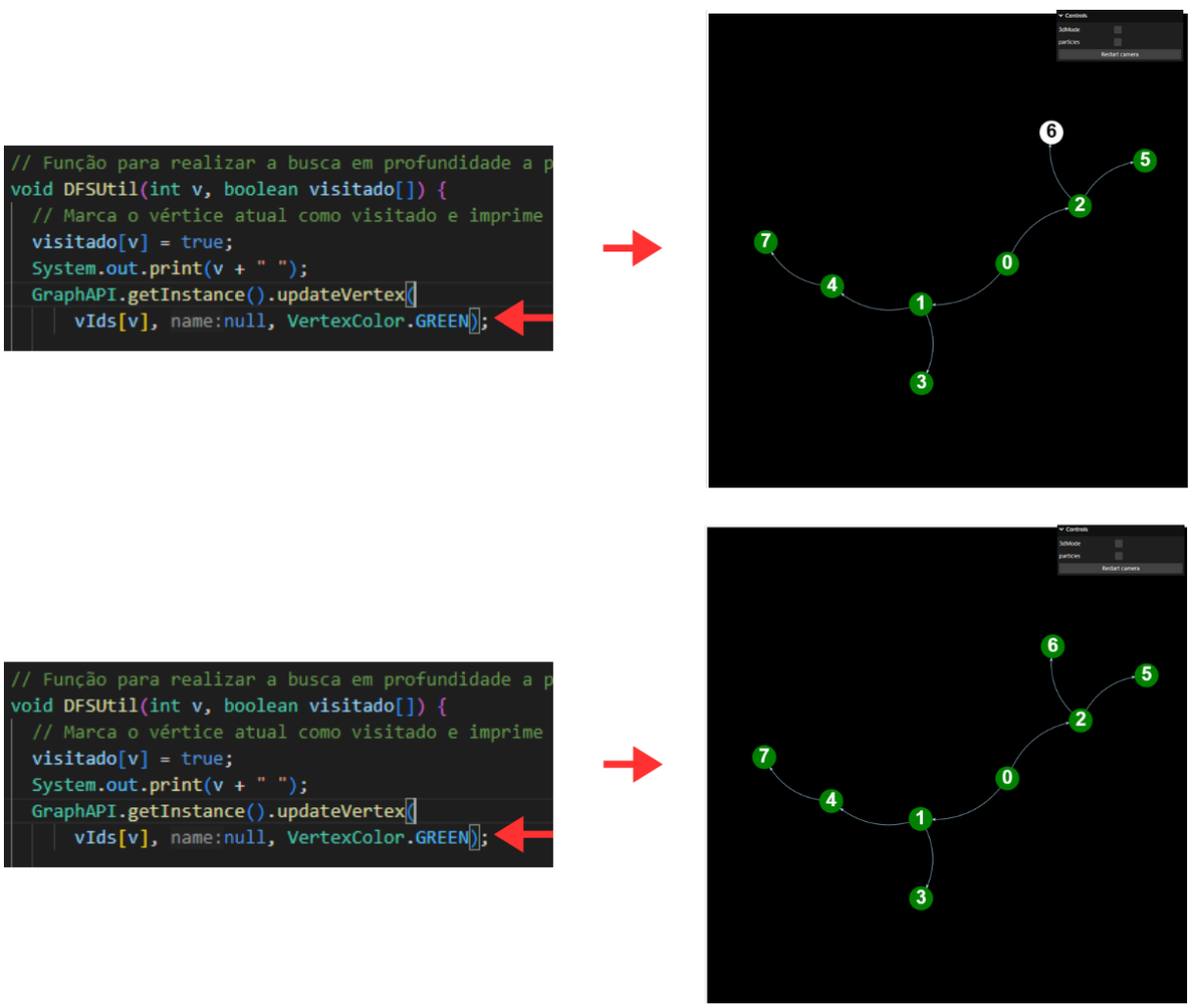

Figura 5.12: DFT - Árvore percorrida

6 Conclusão

A implementação desenvolvida representa um avanço significativo na análise dinâmica de código com representações visuais. A capacidade de observar o comportamento das estruturas de dados durante a execução do programa proporciona *insights* valiosos sobre a interação das partes do *software* entre si. Essa abordagem não apenas facilita a identificação de problemas e otimização de desempenho, mas também enriquece a compreensão do comportamento real do programa em diferentes cenários de uso.

O uso de visualizações interativas oferece uma abordagem abrangente e prática para o estudo e desenvolvimento de *software*, contribuindo para a melhoria da eficiência, qualidade e robustez dos sistemas desenvolvidos. Além disso, a integração de visualizações interativas como ferramenta educacional pode contribuir significativamente para a compreensão e aplicação prática de conceitos complexos na área de Ciência da Computação.

A aplicação desenvolvida atende a proposta, favorecendo a análise dinâmica de código e permitindo que qualquer usuário, principalmente o iniciante em programação, possa utilizar a ferramenta para avaliar algoritmos desenvolvidos no contexto profissional ou acadêmico.

6.1 Possíveis melhorias

Os seguintes requisitos adicionais podem ser desenvolvidos nesta aplicação, a fim de obter maior usabilidade e melhor experiência de usuário:

- Adicionar autenticação e controle de acesso, tornando a aplicação multiusuário e segregando ambientes de teste. Com isso diversos usuários poderiam utilizar o mesmo ambiente;
- Expandir classe de conexão para outras linguagens;
- Adicionar botão para baixar classes/bibliotecas de conexão e adicionar o *token* referente àquele ambiente de testes automaticamente;

- Adicionar recurso de "linha do tempo", similar ao mecanismo de voltar e avançar do navegador *web*, para memorizar o passo a passo de criação do grafo, permitindo navegar pela execução de um determinado código sem necessariamente habilitar o modo de depuração (*debug*) do mesmo;

- Memorizar posição prévia dos elementos, de forma que novas requisições não reiniciem a disposição do grafo gerado;

- Possibilidade de substituir a classe de conexão por aplicações paralelas para monitorar a aplicação a ser testada. Como exemplo, o uso de Java agents, aliado ao uso de *Annotations* para designar classes e métodos a serem mapeados. Importante salientar que tais recursos podem não ser adequados para iniciantes na programação;

- Definir uma representação gráfica específica, que indique tratamento de exceção (*catch*) e vazamentos de memória;

- Habilitar recursos visuais em arestas, indicando acesso aos dados de uma estrutura (objeto) por outra;

- Exibir legenda com número de nós e arestas, indicando a complexidade da estrutura.

Importante destacar que análises de performance devem ser realizadar no sistema para uso em estruturas de dados mais robustas.

Bibliografia

ASHFAQ, Q.; KHAN, R.; FAROOQ, S. A comparative analysis of static code analysis tools that check java code adherence to java coding standards. In: *2019 2nd International Conference on Communication, Computing and Digital systems (C-CODE)*. [S.l.: s.n.], 2019. p. 98–103.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao teste de software*. [S.l.]: Elsevier, 2007.

DROZDEK, A. *Data Structures and Algorithms in C++*. Cengage Learning, 2012. ISBN 9781285415017. Disponível em: ⟨https://books.google.com.br/books?id=PRgLAAAAQBAJ⟩.

FUCHS, J. et al. Treeducation: A visual education platform for teaching treemap layout algorithms. *IEEE Transactions on Visualization and Computer Graphics*, p. 1–16, 2024.

GHANDGE, A. B. et al. Algoassist: Algorithm visualizer and coding platform for remote classroom learning. In: *2021 5th International Conference on Computer, Communication and Signal Processing (ICCCSP)*. [S.l.: s.n.], 2021. p. 1–6.

GUPTA, A. S.; VYAWAHARE, M. Algoviz : Algorithm visualization. In: *2023 5th Biennial International Conference on Nascent Technologies in Engineering (ICNTE)*. [S.l.: s.n.], 2023. p. 1–5.

HUANG, C.-Y. et al. Code coverage measurement for android dynamic analysis tools. In: *2015 IEEE International Conference on Mobile Services*. [S.l.: s.n.], 2015. p. 209–216.

LIN, J.; ZHANG, H. Data structure visualization on the web. In: *2020 IEEE International Conference on Big Data (Big Data)*. [S.l.: s.n.], 2020. p. 3272–3279.

PERHÁC, P.; SIMONÁK, S. Interactive system for algorithm and data structure visualization. *Computer Science Journal of Moldova*, v. 88, n. 1, p. 28–48, 2022.

PRABHAKAR, G. et al. Analysis of algorithm visualizer to enhance academic learning. In: *2022 2nd International Conference on Innovative Practices in Technology and Management (ICIPTM)*. [S.l.: s.n.], 2022. v. 2, p. 279–282.

SULLIVAN, M.; CHILLAREGE, R. Software defects and their impact on system availability-a study of field failures in operating systems. In: *[1991] Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium*. [S.l.: s.n.], 1991. p. 2–9.

TRIVEDI, A. et al. Algorhythm - a sorting and path-finding visualizer tool to improve existing algorithms teaching methodologies. In: *2023 13th International Conference on Cloud Computing, Data Science Engineering (Confluence)*. [S.l.: s.n.], 2023. p. 158–169.