UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

LEANDRO RODRIGUES ROCHA
Orientador: Prof. Dr. Reinaldo Silva Fortes
Coorientador: Prof. Dr. Pedro Henrique Lopes Silva

# APRIMORAMENTOS NO BACK-END DO OPCODERS JUDGE

Ouro Preto, MG
2025

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

LEANDRO RODRIGUES ROCHA

# APRIMORAMENTOS NO BACK-END DO OPCODERS JUDGE

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

**Orientador:** Prof. Dr. Reinaldo Silva Fortes
**Coorientador:** Prof. Dr. Pedro Henrique Lopes Silva

Ouro Preto, MG
2025

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

# FOLHA DE APROVAÇÃO

**Leandro Rodrigues Rocha**

**Aprimoramentos no back-end do opCoders Judge**

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 7 de Abril de 2025.

Membros da banca

Reinaldo Silva Fortes (Orientador) - Doutor - Universidade Federal de Ouro Preto
Pedro Henrique Lopes Silva (Coorientador) - Doutor - Universidade Federal de Ouro Preto
Rodrigo Geraldo Ribeiro (Examinador) - Doutor - Universidade Federal de Ouro Preto
Guilherme Augusto Anício Drummond do Nascimento (Examinador) - Bacharel - Universidade Federal de Ouro Preto

Reinaldo Silva Fortes, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 7/04/2025.

Documento assinado eletronicamente por **Reinaldo Silva Fortes**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 08/04/2025, às 09:21, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0886334** e o código CRC **511777B9**.



R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

# Agradecimentos

Agradeço, primeiramente, a Deus por Sua orientação e bênçãos durante esta jornada. À minha família e amigos, pelo suporte e apoio ao longo do curso. Ao meu orientador, Reinaldo Fortes, e ao meu coorientador, Pedro Silva, pela orientação e pelo valioso conhecimento que compartilharam, essenciais para o desenvolvimento deste trabalho. Por fim, agradeço a Universidade Federal de Ouro Preto e a todos os professores que contribuíram para a minha formação.

# Resumo

As disciplinas de algoritmos e programação desempenham um papel fundamental na formação de estudantes nas áreas de exatas. No entanto, muitos enfrentam dificuldades devido à complexidade abstrata desses conteúdos. Visando auxiliar no aprendizado dos estudantes, foi desenvolvida a plataforma *opCoders Judge*, um Juiz automático para correção de códigos-fonte, fornecendo aos alunos notas e *feedbacks* detalhados sobre suas respostas. Embora a plataforma tenha sido aprimorada ao longo do tempo, ainda há deficiências a serem melhoradas na parte de correção dos códigos-fonte. Este trabalho aborda tais melhorias, com foco em refatorar o código já existente para que o mesmo siga um padrão e fique bem estruturado, facilitando implementações futuras e com foco também nas análises realizadas durante a etapa de correção, visando tornar o corretor mais abrangente e contribuir ainda mais para o aprendizado dos alunos.

**Palavras-chave**: Algoritmos. Corretor Automático. Aprimoramento.

# Abstract

The disciplines of algorithms and programming play a fundamental role in students' education in exact sciences. However, many face difficulties due to the abstract complexity of these contents. With the aim of assisting students' learning, the platform "opCoders Judge" was developed in 2019, an automatic Judge for correcting source codes, providing students with detailed grades and feedback on their answers. Although the platform has been improved over time, there are still deficiencies to be addressed in the correction of source codes. This work addresses such improvements, focusing on refactoring the existing code to follow a standard and be well-structured, facilitating future implementations. Additionally, it focuses on the analyses performed during the correction stage, aiming to make the corrector more comprehensive and further contribute to students' learning.

**Keywords**: Algorithms. Automatic corrector. Enhancement.

# Lista de Ilustrações

# Lista de Tabelas

# Lista de Abreviaturas e Siglas

| | |
|---|---|
| AST | *Abstract Syntax Trees* |
| CFG | *Control-Flow Graph* |
| DECOM | Departamento de Computação |
| JSON | *JavaScript Object Notation* |
| PEP8 | Python *Enhancement Proposals 8* |
| POO | Programação Orientada a Objetos |
| SAM | *Self-Assessment Manikin* |
| SQL | *Structured Query Language* |
| SUS | *System Usability Scale* |
| UAT | *User Acceptance Testing* |
| UFOP | Universidade Federal de Ouro Preto |

# Sumário

# 1 Introdução

As disciplinas de Algoritmos e Programação são componentes essenciais em diversos cursos de graduação, especialmente na área de ciências exatas. Seu propósito é cultivar o raciocínio lógico necessário para a jornada acadêmica (MARCUSSI et al., 2016). Contudo, muitos estudantes enfrentam desafios consideráveis ao lidar com essas matérias, em grande parte devido à complexidade abstrata que exigem. Devido à elevada quantidade de estudantes, a capacidade dos professores em fornecer assistência e analisar os códigos de todos os alunos de maneira rápida é limitada, o que acaba por dificultar o processo de aprendizado. Como resposta a essa necessidade, foi concebido o sistema de avaliação de exercícios de programação denominado *opCoders Judge*, ou seja, uma tutoria automatizada. Esse mecanismo permite que os alunos submetam suas atividades e obtenham avaliações em tempo hábil, facilitando o processo de aprendizagem.

A tutoria automatizada representa uma abordagem inovadora destinada a capacitar os alunos a desenvolverem soluções para exercícios, sem depender intimamente da orientação direta de um professor humano. Seu foco reside na oferta de orientações úteis para soluções que apresentem erros ou estejam incompletas, indo além de mensagens simplificadas, como “erro na linha X” (BRITO, 2019).

O corretor *opCoders Judge* foi desenvolvido em 2019 por Brito (2019). No início, uma versão offline do corretor foi implementada para avaliar os exercícios dos alunos. As tarefas eram gerenciadas pelo *Moodle*, onde os alunos tinham acesso aos enunciados em formato PDF e enviavam suas soluções como arquivos de código-fonte. Esses arquivos eram então baixados para a máquina local do professor e corrigidos automaticamente. As respostas aos alunos eram enviadas como arquivos PDF contendo os resultados da avaliação. Posteriormente, foi desenvolvida uma versão web do corretor desenvolvido por Patrocínio (2023). Nessa versão, os alunos podiam visualizar os enunciados em formato de página web, enviar seus códigos-fonte e visualizar os resultados. Uma vantagem significativa dessa versão foi a mudança para uma correção online, onde os alunos tiveram acesso aos resultados da correção automática em poucos segundos após o envio de suas respostas. Entretanto, o corretor ainda está em fase de desenvolvimento e requer melhorias na análise de códigos-fonte, conforme avaliação de usabilidade feita por Cedraz (2023).

Para fazer a avaliação de um código-fonte, duas análises principais podem ser consideradas: a **estática** e a **dinâmica**. No âmbito deste estudo, concentrou-se no aprimoramento da análise dinâmica. A análise estática se direciona à avaliação do código submetido pelo aluno. O objetivo principal é identificar a presença de estruturas, comandos, funções e outros requisitos delineados no enunciado da questão. Essa abordagem não se limita à mera verificação da correção da resposta final, ela busca também avaliar a abordagem adotada e a metodologia empregada na

sua elaboração. Esta análise é crucial, uma vez que frequentemente um exercício pode ser resolvido de maneira que diverge dos requisitos impostos pelo enunciado. Tome como exemplo um cenário onde o exercício estipula a criação de uma função para realizar determinadas operações, nesse contexto, uma solução alternativa poderia gerar a resposta correta sem a necessidade de utilizar uma função, mas essa abordagem não estaria alinhada com a ideia proposta pelo exercício. Em determinadas disciplinas, a utilização dos recursos da linguagem de programação pode ser restrita, visando possibilitar que o aluno consiga implementar e compreender integralmente os fundamentos lógicos subjacentes a certos comandos internos da linguagem. A análise estática também é aplicada nestes casos, verificando se o código foi elaborado exclusivamente mediante o emprego de recursos autorizados.

Já a análise dinâmica é uma técnica utilizada na engenharia de software para examinar o comportamento do programa durante sua execução. Em contraste com a análise estática, que examina o código-fonte sem executá-lo, a análise dinâmica envolve a execução real do programa, permitindo a observação do fluxo de execução, a identificação de erros e a avaliação do desempenho em tempo de execução. Uma aplicação relevante da análise dinâmica são os testes unitários, que servem para testar partes específicas do código isoladamente, garantindo que produzam os resultados esperados. Por exemplo, consideremos uma função em um código que realiza a soma de dois números. Podemos empregar a análise dinâmica para verificar se essa função executa os cálculos corretamente, ao executá-la com parâmetros de teste e comparar o resultado retornado com o valor esperado.

As próximas subseções destacam a importância e os objetivos planejados para este trabalho. A Seção 1.1 apresenta a motivação e relevância do presente estudo. A Seção 1.2 mostra os objetivos que visam ser alcançados na realização do trabalho. Por fim, na Seção 1.3 é apresentada a estrutura e como o trabalho foi organizado.

## 1.1 Justificativa

A evolução do corretor automático *opCoders Judge* tem sido notável desde sua concepção. Inicialmente simples, a plataforma foi aprimorada por outros estudantes, com foco principalmente no desenvolvimento da interface. No entanto, a parte da correção de código-fonte não acompanhou o mesmo ritmo de evolução e ainda apresenta áreas que necessitam de melhorias, especialmente nas análises realizadas pelo corretor. Além disso, foi observado que o código, escrito na linguagem *Python*, não seguia algumas diretrizes e não estava devidamente arquitetado. Diante desse cenário, a relevância e a necessidade de aprofundar a investigação neste estudo são justificadas pelas melhorias que precisam ser implementadas.

## 1.2 Objetivos Gerais e Específicos

O objetivo geral deste estudo é abordar as deficiências na correção de códigos, especialmente nas análises estáticas e dinâmicas, realizando a refatoração e implementação com o propósito de estruturar o código de maneira eficiente, tornando-o mais apto a futuras expansões. São objetivos específicos:

- Refatorar o código do módulo de correção automática *opCoders Judge* para ficar consoante a PEP8. [1]
- Refatorar estrutura de classes do módulo de correção automática *opCoders Judge*.
- Implementar melhorias na análise dinâmica feita pelo *opCoders Judge*, adicionando informações de erros contidos na saída do usuário que não penalizaram a nota obtida e fazendo a diferenciação de erros que penalizaram.
- Implementar melhorias na análise dinâmica feita pelo *opCoders Judge*, aprimorando o tratamento de casos com linhas em banco erradas na saída do aluno.
- Implementar melhorias na análise dinâmica feita pelo *opCoders Judge*, adicionando a nota individual de cada critério na saída do corretor.
- Realizar testes qualitativos das melhorias nas avaliações feitas.

## 1.3 Organização do Trabalho

Na sequência desse trabalho, serão abordados os capítulos de Revisão Bibliográfica, Capítulo 2, onde serão discutidos conceitos-chave e trabalhos relacionados. Em seguida, temos o Desenvolvimento, Capítulo 3, que inclui os métodos utilizados. No Capítulo 4, são apresentados e discutidos os resultados obtidos. E por fim, Considerações Finais, Capítulo 5, destacando implicações e sugestões para pesquisas futuras.

[1] <https://peps.python.org/pep-0008/>

# 2 Revisão Bibliográfica

Neste capítulo, serão apresentados os principais conceitos relacionados aos temas discutidos, os quais foram abordados na Seção 2.1. Além disso, serão explorados os trabalhos relacionados ao presente estudo, os quais foram apresentados na Seção 2.2.

## 2.1 Fundamentação Teórica

A seguir, são apresentados conceitos importantes para o entendimento geral do trabalho. Na Seção 2.1.1, serão discutidos os conceitos de corretores automáticos de código. Na Seção 2.1.2, será abordada a análise estática. Na Seção 2.1.3, será abordada a análise dinâmica. Por fim, na Seção 2.1.4, será explorado como as análises estáticas e dinâmicas podem se complementar.

### 2.1.1 Corretores automáticos de código

Os corretores automáticos de código possuem como objetivo validar a consistência de um algoritmo por meio de vários casos de teste. Este processo é semelhante ao utilizado pela engenharia de software no teste da caixa preta ou funcional. No teste da caixa preta não se tem conhecimento interno do algoritmo, os dados de entrada são fornecidos, a partir deles o algoritmo interpreta os dados recebidos e o resultado de saída gerado pelo algoritmo é comparado com o esperado (NAZÁRIO; SOUZA, 2010). O tipo de análise feita nesses casos é a análise dinâmica, que será explicada na Seção 2.1.3. Existem várias plataformas online para a prática de programação que têm o funcionamento semelhante ao citado acima, dentre elas pode-se citar algumas como: *Codeforces*[1], *Beecrowd*[2], *Neps Academy*[3], *AtCoder*[4] e o *leetCode*[5].

Na maioria dessas plataformas, o *feedback* recebido pelo usuário é uma pontuação baseada na quantidade de casos de teste que receberam resposta correta e a eficiência do código com relação ao tempo de execução e quantidade de memória utilizada, como pode ser visualizado na Figura 2.1, que mostra exemplos de respostas retornadas após a submissão de alguns problemas. A resposta *Accepted* indica que o código submetido passou em todos os casos de teste. Já a resposta *Wrong answer* indica que o código falhou em alguns casos de teste, mostrando a porcentagem de acerto entre parênteses. Quando a resposta é *Time limit exceeded*, significa que o tempo de execução do código excedeu o limite permitido na questão. Por fim, a resposta *Memory limit exceeded* indica que o código ultrapassou a quantidade de armazenamento de memória permitido na questão.

[1] Disponível em <https://codeforces.com/>. Acessado em fevereiro de 2024.
[2] Disponível em <https://www.beecrowd.com.br/judge/en/login>. Acessado em fevereiro de 2024.
[3] Disponível em <https://neps.academy/br>. Acessado em fevereiro de 2024.
[4] Disponível em <https://atcoder.jp/>. Acessado em fevereiro de 2024.
[5] Disponível em <https://leetcode.com/>. Acessado em fevereiro de 2024.

Figura 2.1 – Exemplo dos tipos de *feedback* recebidos na plataforma *Beecrowd*.

| # | | PROBLEMA | RESPOSTA | LINGUAGEM | HORA | DATA |
|---|---|---|---|---|---|---|
| 32834537 | 2433 | Vende-se | Accepted | C++17 | 0.056 | 08/04/2023 13:14 |
| 32738265 | 1063 | Trilhos Novamente... Traç... | Accepted | C++17 | 0.132 | 03/04/2023 21:55 |
| 25122167 | 1255 | Frequência de Letras | Accepted | Python 3.8 | 0.327 | 29/10/2021 22:20 |
| 25122159 | 1255 | Frequência de Letras | Wrong answer (65%) | Python 3.8 | 0.461 | 29/10/2021 22:19 |
| 25121881 | 1255 | Frequência de Letras | Accepted | Python 3.8 | 0.362 | 29/10/2021 21:49 |
| 25120276 | 1159 | Soma de Pares Consecutivos | Accepted | Python 3.8 | 0.062 | 29/10/2021 19:43 |
| 25042731 | 1425 | Presente?! | Accepted | Python 3.8 | 0.135 | 24/10/2021 23:33 |
| 25042612 | 1425 | Presente?! | Accepted | C++ | 0.000 | 24/10/2021 23:20 |
| 25042453 | 1425 | Presente?! | Memory limit exceeded | C++ | 0.569 | 24/10/2021 22:59 |
| 25027978 | 1968 | A Terra Desconhecida | Accepted | C++ | 0.785 | 23/10/2021 13:14 |
| 25027801 | 1968 | A Terra Desconhecida | Time limit exceeded | C++ | 2.000 | 23/10/2021 13:00 |
| 25027733 | 1968 | A Terra Desconhecida | Memory limit exceeded | C++ | 1.067 | 23/10/2021 12:56 |
| 25023250 | 1968 | A Terra Desconhecida | Time limit exceeded | C++ | 2.000 | 23/10/2021 01:49 |

Fonte: (BEECROWD, 2024).

No entanto, no contexto do ensino de programação, é importante identificar como o código foi escrito e quais estruturas e funcionalidades da linguagem foram utilizadas. Para transcender a ideia convencional de corretores e adicionar esse tipo de verificação, pode-se utilizar a análise estática, que será explicada na próxima Seção 2.1.2.

### 2.1.2 Análise Estática

Segundo Louridas (2006), a análise estática de código é a revisão do código-fonte por meio da busca por padrões conhecidos de erros, ou erros previsíveis em uma aplicação sem que seja necessário executá-la.

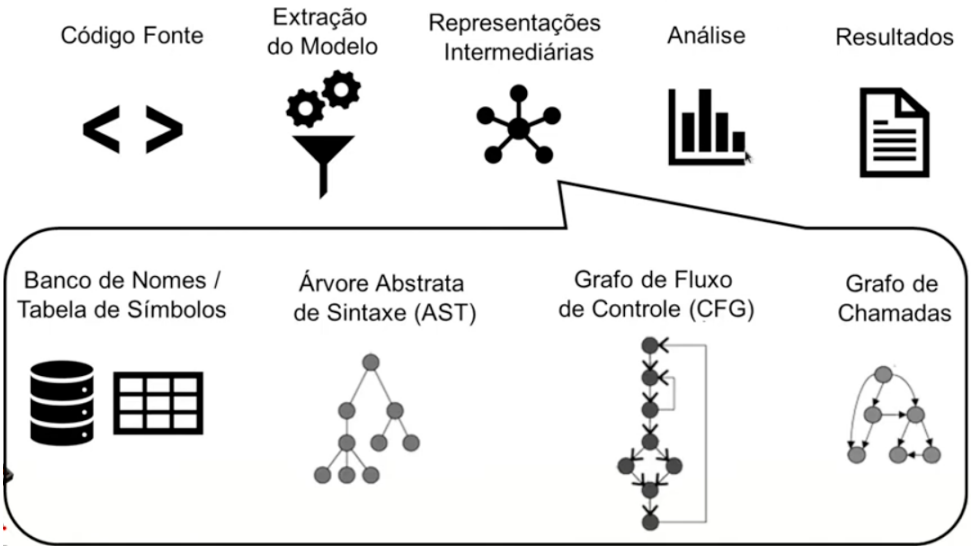
Os analisadores estáticos podem examinar o código-fonte, o código de bytes ou os binários de um programa. Embora cada verificador de código opere de forma distinta, a maioria deles apresenta características fundamentais em comum. Os verificadores estáticos examinam o programa e criam uma representação abstrata, utilizando-a para identificar padrões de erro (LOURIDAS, 2006). O fluxo realizado em uma análise estática e possíveis representações abstratas, também conhecidas como representações intermediárias, pode ser visualizado na Figura 2.2. Existem vários tipos de representações abstratas que podem ser utilizadas:

- Árvores de Sintaxe Abstrata (AST, do inglês *Abstract Syntax Trees*): Uma AST representa a estrutura hierárquica de um programa sem incluir todos os detalhes da sintaxe. É uma representação mais simples e abstrata do código-fonte.
- Grafos de Fluxo de Controle (CFG, do inglês *Control-Flow Graph*): Um CFG é uma representação gráfica do fluxo de controle em um programa. Cada nó no grafo representa um bloco de código e as arestas indicam as possíveis transições entre os blocos.
- Grafos de Dependência: Esses grafos representam as dependências entre diferentes partes de um programa, como dependências de dados, dependências de controle e dependências

entre módulos.

- Representações Baseadas em Lógica: Essas representações usam lógica matemática para descrever propriedades e comportamentos do programa de forma abstrata.
- Modelos de Estado: Esses modelos descrevem o comportamento do programa em termos de estados e transições de estado.
- Álgebra de Programas: Uma abordagem baseada em álgebra para representar programas e suas transformações, onde as operações algébricas são usadas para analisar e transformar o código-fonte.

Figura 2.2 – Fluxo da Análise Estática de um código.




Fonte: (NTNU, 2020).

Cada analisador estático mantém um conjunto de vulnerabilidades em seu banco de dados, que são examinadas no código; a maioria dos produtos oferece a capacidade de incluir regras personalizadas. A análise estática pode ser aplicada a módulos ou a códigos em desenvolvimento, embora seja importante notar que quanto mais abrangente o código, mais abrangente e precisa será a análise realizada. Em contrapartida, os testes demandam casos de teste ou dados de entrada, além de exigirem artefatos suficientemente completos para serem executáveis, potencialmente incluindo *drivers* de suporte, *stubs* ou componentes simulados. Logo, em certas situações, é mais prático empregar verificadores de código estático durante o desenvolvimento, mas é importante salientar que a análise estática de código não substitui, mas sim complementa o processo de teste, pois testes têm a vantagem de possivelmente revelar falhas completamente inesperadas que não são identificadas pela análise estática.

Existem diversas ferramentas no mercado que realizam a análise estática de código, alguns exemplos são: *SonarQube* [6], que analisa códigos *Python*, *FindBugs* [7], que analisa códigos Java e a ferramenta *Splint* / *LCLint* [8], que serve para análise de códigos escritos em C. Porém, o foco da maioria delas é encontrar possíveis *bugs* e vulnerabilidades de segurança.

### 2.1.3 Análise Dinâmica

Na análise dinâmica, o algoritmo é executado com entradas pré-definidas e suas saídas são comparadas com as saídas esperadas; é um processo de avaliação de um programa de computador em tempo de execução. Ao contrário da análise estática, que examina o código-fonte sem executá-lo, a análise dinâmica envolve a execução real do programa para identificar erros, comportamentos inesperados, falhas de segurança e outras questões durante a execução (BALL, 1999).

A análise dinâmica é usualmente utilizada para realizar diversos tipos de testes de *softwares*, sendo eles:

- Testes de Unidade: São testes que verificam unidades individuais de código, como funções ou métodos, para garantir que eles funcionem conforme o esperado.
- Testes de Integração: Testes que verificam a interação entre diferentes unidades ou módulos de código para garantir que eles funcionem corretamente quando combinados.
- Testes de Sistema: Testes que avaliam o sistema na totalidade para garantir que todos os componentes integrados funcionem conforme os requisitos.
- Testes de Aceitação do Usuário (UAT, do inglês *User Acceptance Testing*): Testes conduzidos pelos usuários finais para verificar se o sistema atende aos requisitos e expectativas do usuário.
- Testes de Desempenho: Avaliação do desempenho do software em condições específicas, como carga de trabalho elevada, para garantir que ele atenda aos requisitos de desempenho.
- Testes de Segurança: Testes que buscam identificar vulnerabilidades de segurança no software.
- Testes de Regressão: Testes realizados para garantir que novas alterações no código não introduzam regressões, ou seja, não quebrem funcionalidades existentes.
- Testes de Estresse: Avaliação do comportamento do sistema sob condições extremas, como carga máxima, para verificar sua robustez e comportamento sob pressão.

[6] Disponível em <https://www.sonarsource.com/products/sonarqube/>. Acessado em fevereiro de 2024.
[7] Disponível em <https://findbugs.sourceforge.net/>. Acessado em fevereiro de 2024.
[8] Disponível em <https://splint.org/>. Acessado em fevereiro de 2024.

### 2.1.4 Análise Estática x Análise Dinâmica

Como descrito nas sessões anteriores, a análise estática se concentra na revisão do próprio código-fonte, enquanto a análise dinâmica verifica a execução do mesmo. Mas vale ressaltar também que as análises estáticas e dinâmicas são técnicas complementares em diversas dimensões, sendo elas: **completude**, **escopo** e **precisão**. Descritas a seguir:

- **Completude**: Em geral, as análises dinâmicas geram "invariantes de programa dinâmico", propriedades verdadeiras para o conjunto observado de execuções. A análise estática pode ajudar a determinar se essas "invariantes" dinâmicas são realmente invariantes em todas as execuções do programa. Nos casos em que as análises dinâmica e estática discordam, existem duas possibilidades: (1) a análise dinâmica está errada porque não abrangeu um número suficiente de execuções; (2) a análise estática está errada porque analisou caminhos inviáveis (caminhos que nunca podem ser executados). Como a análise dinâmica examina as execuções reais do programa, ela não sofre do problema de caminhos inviáveis que podem prejudicar as análises estáticas. Por outro lado, a análise dinâmica, por definição, considera menos caminhos de execução do que a análise estática.

- **Escopo**: Como a análise dinâmica examina um caminho de programa muito longo, ela tem o potencial de descobrir dependências semânticas entre entidades de programa amplamente separadas no caminho (e no tempo). A análise estática normalmente é restrita ao escopo de um programa que pode analisar de forma eficaz e eficiente e pode ter dificuldade em descobrir tais "dependências à distância".

- **Precisão**: A análise dinâmica tem a vantagem de examinar o domínio concreto da execução do programa. A análise estática deve abstrair este domínio para garantir o encerramento da análise, perdendo assim informações desde o início. A abstração pode ser uma técnica útil para reduzir a sobrecarga do tempo de execução da análise dinâmica e reduzir a quantidade de informações registradas, mas não é necessária para o encerramento.

Portanto, evidencia-se a importância tanto da análise estática quanto da análise dinâmica. Embora distintas, ambas podem complementar-se mutuamente, permitindo uma avaliação mais abrangente e precisa do código-fonte.

### 2.1.5 Similaridade de *strings*

A comparação entre cadeias de caracteres (*strings*) é um problema relevante em diversas áreas, como recuperação de informação, processamento de linguagem natural e análise de dados textuais. No contexto deste trabalho, a similaridade entre *strings* é utilizada para avaliar automaticamente a correção de respostas de alunos em relação a uma resposta esperada em exercícios de programação.

Uma abordagem amplamente utilizada para medir o grau de similaridade entre duas *strings* é a similaridade baseada na distância de edição, que quantifica o número mínimo de operações necessárias para transformar uma *string* em outra. Dentre as métricas de distância de edição, destaca-se a distância de Levenshtein, que considera inserções, deleções e substituições de caracteres (LISBACH; MEYER, 2013) . Essa métrica permite quantificar o grau de diferença entre a resposta do aluno e a resposta esperada, servindo como base para o cálculo da pontuação automática no corretor.

Além disso, algoritmos de *Fuzzy Matching* permitem comparar *strings* aproximadas, sendo úteis em cenários onde pequenas variações podem ocorrer. A função `partial_ratio` [9], utilizada neste trabalho, identifica a melhor correspondência parcial entre as *strings* analisadas, tornando a comparação mais flexível ao desconsiderar prefixos ou sufixos desnecessários. Dessa forma, substrings de diferentes tamanhos podem ser comparadas de maneira mais adaptativa, resultando em um valor percentual que representa o grau de similaridade entre a resposta do aluno e a solução correta.

Essa técnica é essencial para a avaliação automatizada, pois permite que o corretor não seja excessivamente rígido com pequenas variações na resposta dos alunos, tornando a correção mais flexível e justa. Além disso, a similaridade aproximada entre *strings* pode ser combinada com outras heurísticas para aprimorar ainda mais a precisão do sistema de correção automática.

## 2.2 Trabalhos Relacionados

Na literatura, é possível identificar alguns estudos e documentos acadêmicos como os de Brito (2019), Brito e Fortes (2019), Patrocínio (2023), Cedraz (2023) e Mendonça (2023). Essas fontes apresentam a concepção original da plataforma e descrevem as melhorias e desenvolvimentos implementados ao longo do tempo em seu sistema de correção. Os demais estudos mencionados dizem respeito à aplicação de técnicas e ao uso de outros juízes online no ambiente de aprendizagem.

A proposta inicial foi desenvolvida por Brito (2019), que aborda uma metodologia de ensino que utiliza um corretor automático de códigos e exploração de questões objetivas para consolidar os assuntos abordados durante as aulas teóricas. O corretor foi inicialmente desenvolvido para funcionar de forma *offline* e não tinha uma interface gráfica para interação do usuário. Uma das conclusões do trabalho foi que, do ponto de vista pedagógico, aplicar apenas análise estática ou análise dinâmica para a correção de um programa não é interessante, visto que pode penalizar consideravelmente um aluno que desenvolveu um raciocínio lógico próximo do que era esperado pelo professor. A estratégia proposta é gerar o resultado para o aluno a partir de ponderações dos resultados obtidos pela análise estática e dinâmica que mais se aproxime de

[9] Disponível em <https://github.com/seatgeek/thefuzz?tab=readme-ov-file>. Acessado em Janeiro de 2025.

uma avaliação manual realizada pelo professor. O presente trabalho visa aprimorar ainda mais esse processo.

No trabalho de Patrocínio (2023), um aprimoramento e finalização da versão *online* para o corretor foram conduzidos visando fornecer a correção da atividade submetida pelo aluno em tempo reduzido e auxiliar o processo de gestão e correção de atividades pelos professores. Além disso, a implementação realizada no trabalho possibilitou a inserção de novas linguagens de programação por parte da interface, que suportava apenas a linguagem *Python*, possibilitando, assim, que o corretor pudesse atender a outras disciplinas que envolvem a prática de programação de computadores.

A monografia de Cedraz (2023) se concentra em entender se a ferramenta oferece aos usuários uma interface fácil de usar e que possibilite que os mesmos concluam as suas tarefas de forma satisfatória e eficiente. Dois métodos conhecidos na literatura de Interação Humano-Computador foram utilizados para fazer a análise de usabilidade: a avaliação por inspeção, por meio das 10 Heurísticas de Nielsen (NIELSEN, 1994), e a avaliação por investigação, a partir da aplicação dos questionários *System Usability Scale* (SUS) (BROOKE, 1996) e *Self-Assessment Manikin* (SAM) (LANG, 1980). A análise por inspeção revelou sérios problemas de usabilidade na interface da ferramenta, o que levou à criação de um protótipo de alta fidelidade para abordar as questões identificadas. Este protótipo demonstrou uma notável melhoria na usabilidade.

A pesquisa conduzida por Mendonça (2023) concentra-se na criação de um módulo dedicado à geração de um amplo espectro de questões exclusivas. Foi desenvolvido um método para que o professor elabore um modelo de questão e, a partir deste modelo, variadas versões de questões podem ser criadas pelo sistema. As questões-modelo possuem metadados que auxiliam na sua classificação, como a área de conhecimento, nível de dificuldade da questão, tópicos relacionados ao ensino de programação e *tags* que permitem ao professor estabelecer um padrão de características aplicáveis às questões. Para obter outras versões de questões, a partir da questão-modelo elaborada pelo professor, são utilizadas variáveis do texto que podem receber diversos valores; logo, quanto maior o número de variáveis no texto, maior será a diversidade de elementos em uma versão da questão. Essa iniciativa visa aprimorar a plataforma e oferecer suporte aos professores na criação de tarefas dinâmicas a partir de um modelo.

No estudo de Striewe e Goedicke (2014), uma abordagem de análise estática é apresentada, focalizando o ambiente de aprendizagem. O propósito é estabelecer uma correlação entre os resultados técnicos da análise do código-fonte e os benefícios didáticos derivados dela para o ensino de programação e a geração de *feedback*. No decorrer das comparações de abordagens delineadas no estudo, destaca-se a consideração entre avaliar um único arquivo em contraste com a avaliação de vários arquivos de código. A análise revela que, em situações em que o código é composto por diversos arquivos, é crucial que a avaliação leve em conta essa complexidade. Por exemplo, a criação de um método em um arquivo que não é chamado nele mesmo pode erroneamente ser identificada como inútil pela análise, quando, na realidade, esse método pode

estar sendo invocado em outro arquivo. Assim, a conclusão do estudo ressalta a necessidade, em ambientes de aprendizagem, de empregar ferramentas capazes de lidar eficientemente com a análise de múltiplos arquivos fonte.

O trabalho de Cruz et al. (2022) explana a dificuldade dos alunos em aprender programação devido às complexidades e abstrações e propõe um modelo de plano de ensino voltado para auxiliar os professores na adoção de metodologias ativas no processo de aprendizagem. Este plano de ensino é fundamentado na utilização da plataforma *Beecrowd* para maratonas de programação, combinada com o emprego de metodologias ativas, visando melhorar a eficácia do ensino de programação. No trabalho, foram propostas três abordagens para a integração da maratona de programação em disciplinas de algoritmos. Estas incluem: a utilização de questões avulsas, competições de curta duração e competições de longa duração. Para implementar essas metodologias, foi empregada a plataforma *Beecrowd*, que opera com um sistema categorizado de problemas com níveis de dificuldade. Na plataforma *Beecrowd*, os problemas são categorizados em diversas áreas, tais como Iniciante, AD-HOC, Strings, Estruturas e Bibliotecas, Matemática, Paradigmas, Grafos, Geometria Computacional e SQL. Para avaliar a proposta, foram utilizados 6 semestres da disciplina de algoritmos. Os resultados mostraram que, ao ter o auxílio de uma plataforma de correção automática de códigos, o aproveitamento das turmas tende a crescer, fazendo com que as notas mais altas se repitam com mais constância, enquanto que nos períodos em que não foi aplicada a metodologia, ocorreu uma minoração das notas. Por fim, foi concluído que a incorporação dessas metodologias por meio do conceito da maratona de programação é viável e possibilita ao professor criar uma imersão dos alunos em um ambiente totalmente voltado para a prática da programação de computadores.

Uma parte dos trabalhos supracitados apresenta informações sobre a criação e evolução do corretor *opCoders Judge*, porém a ênfase das melhorias se concentrou na parte da interface e usabilidade do sistema. A ideia do presente trabalho é dar continuidade à evolução do corretor proposto, mas com ênfase na correção de códigos-fonte.

# 3 Desenvolvimento

Foi realizada uma análise e compreensão do código atual do corretor. Observou-se que uma parte significativa do código não seguia as boas práticas de codificação; portanto, foram propostas melhorias nesse aspecto. A Seção 3.1 aborda essas melhorias. Na Seção 3.2 será discutido o processo de análise dinâmica realizado pelo corretor, também apresentando as melhorias realizadas.

## 3.1 Refatoração do código

O programa que realiza a correção de códigos-fonte da plataforma *opCoders Judge* foi escrito em linguagem *Python* e passou por várias modificações desde a sua criação até chegar na versão atual. Devido ao fato de não ter sido estabelecido um padrão de implementação, o código não segue a PEP8[1], um conjunto de diretrizes e recomendações para escrever código *Python* de maneira mais clara, legível e consistente. Dentre as diretrizes que não eram seguidas pelo código, destacam-se:

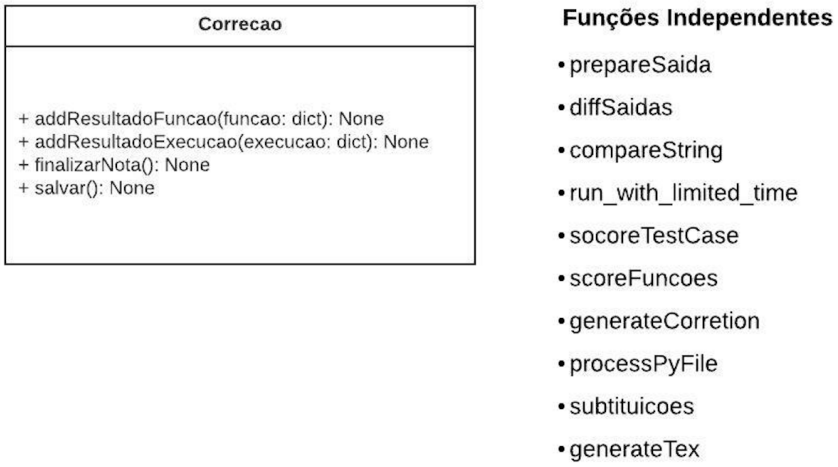
- Uso de 4 espaços por nível de identação: as linhas de continuação devem alinhar os elementos agrupados verticalmente usando a linha implícita do *Python* juntando-se entre parênteses, colchetes e chaves, ou usando um recuo deslocado. Ao usar um recuo deslocado, o seguinte deve ser considerado; não deve haver argumentos na primeira linha e um recuo adicional deve ser usado para distinguir-se claramente como uma linha de continuação
- Limitar todas as linhas a um máximo de 79 caracteres: para blocos de texto longos e fluidos com menos restrições estruturais (documentos ou comentários), o comprimento da linha deve ser limitado a 72 caracteres.
- Linhas em branco: as definições de função e classe de nível superior devem ter duas linhas em branco e as definições de métodos em uma classe são cercadas por uma única linha em branco.
- Comentários: certificar de que haja comentários que sejam claros e facilmente compreensíveis para outros falantes do idioma em que você está escrevendo, além de escrevê-los em idioma inglês.
- Nomes de funções e variáveis: os nomes das variáveis e funções devem estar em letras minúsculas, com palavras separadas por sublinhados conforme necessário para melhorar a legibilidade.

[1] Disponível em: <https://peps.python.org/pep-0008/>. Acessado em fevereiro de 2024.

Todas essas violações de diretrizes dificultavam a leitura e entendimento do código, tornando desafiadora a implementação de novas funcionalidades e a identificação de *bugs*, logo o código foi refatorado considerando todas as regras da PEP8.

O código foi inicialmente elaborado utilizando o paradigma de Programação Orientada a Objetos (POO). Existia uma classe denominada ***Correcao*** com métodos e atributos que processavam e armazenavam o resultado da execução. No entanto, existiam 10 funções que não pertenciam a nenhuma classe. Essas funções desempenhavam operações importantes, tais como a verificação do tempo de execução, a avaliação da similaridade entre as saídas geradas pelo código submetido e a saída correta, a correção de funções, bem como outras operações auxiliares para a correção do código submetido. A Figura 3.1 mostra um diagrama com a classe supracitada e lista todas as funções independentes que havia no código.

Figura 3.1 – Classe de correção antiga e algumas funções independentes.

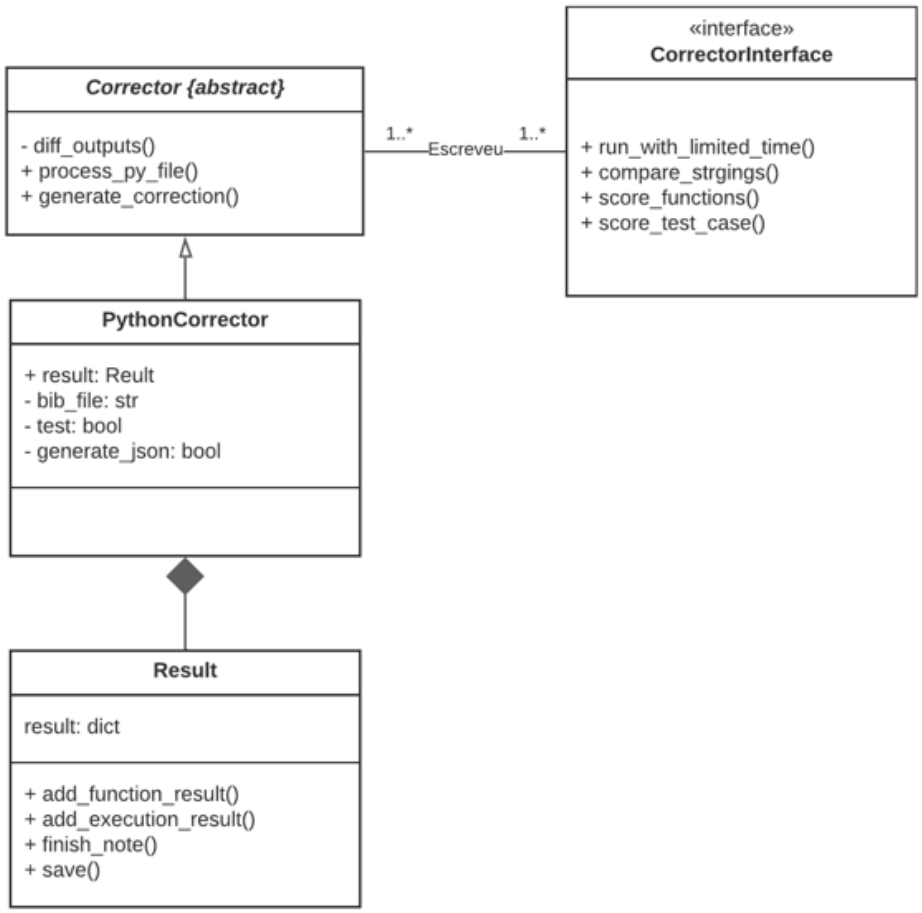



Fonte: Próprio autor.

A nova estrutura de classes tem como objetivo melhorar a organização do código e torná-lo mais propenso à expansão. Isso facilitará a implementação de novas funcionalidades e possíveis futuras expansões para permitir a correção de códigos em outras linguagens de programação, já que atualmente apenas a linguagem *Python* é suportada. A Figura 3.2 apresenta essa nova estrutura, composta por uma interface, cujos métodos serão implementados em uma classe de correção específica para uma linguagem, neste caso, a linguagem *Python*.

O objetivo é tornar genérico o comportamento de um corretor, de tal forma que qualquer outro corretor siga os mesmos padrões e processos. Na prática, a adição de um novo corretor implica somente em herdar e implementar as funções da classe abstrata ***Corrector***.

Figura 3.2 – Novo diagrama de Classes. Autor: Leandro Rocha.




Fonte: Próprio autor.

## 3.2 Melhorias na Análise Dinâmica

A análise dinâmica feita atualmente pelo *opCoders Judge* funciona conforme o fluxo apresentado na Figura 3.3.

Figura 3.3 – Fluxo de execução do corretor.


Fonte: Adaptada de Brito (2019).

Resumidamente, no pré-processamento são definidos casos de teste, compostos por arquivos de entrada, que simulam a entrada de dados de um programa e arquivos de critérios, que definem a saída esperada para cada arquivo de entrada.

Na etapa de correção, o corretor percorre um diretório contendo os arquivos de solução e critérios (um exemplo de arquivo de critérios pode ser visto na Figura 3.4), essa solução

é executada utilizando as entradas pré-definidas e a solução é armazenada em um arquivo temporário. Utiliza-se uma função de comparação para medir o grau de similaridade entre a saída esperada e a saída obtida. Também são feitos alguns procedimentos para evitar penalização na nota do aluno por erros que não estão relacionados diretamente com a lógica implementada. Os procedimentos feitos são:

- Eliminação de linhas em branco.
- Eliminação de acentuação de caracteres.
- Eliminação de caracteres especiais.
- Eliminação de espaços em branco
- Eliminação de pontuação (exceto ponto final)
- Conversão do texto para letras minúsculas.

Figura 3.4 – Exemplo do arquivo de critérios antigo.

```
5
1 1
Informe a quantidade de mulheres: ||0.5000000001
2 1
Informe a quantidade de homens: ||0.5000000001
3 2
15 alunas||2.6666666667666665
preferem refeição vegetariana.||0.3333333334333333
4 2
5 alunos||2.6666666667666665
preferem refeição vegetariana.||0.3333333334333333
5 2
A porcentagem de estudantes que preferem refeição vegetariana é ||0.3333333334333333
15.4%.||2.6666666667666665
```

Fonte: Próprio autor.

E por fim, na geração de saídas, todas as informações referentes à correção são formalizadas e exportadas para um formato de JSON.

As alterações propostas neste trabalho introduzem uma nova abordagem para a correção, que agora é realizada linha a linha sem pré-processar toda a solução do aluno antes. Essa mudança permite diferenciar números de strings, considerar palavras especiais e tratar de forma mais eficiente as linhas em branco.

Um dos desafios na correção de exercícios de programação é atribuir pesos diferentes a determinadas partes da resposta. Embora o corretor antigo desse suporte a essa funcionalidade, ele possuía uma limitação: os números na saída do aluno eram comparados com a saída correta como se fossem strings. Esse método poderia levar a avaliações imprecisas, pois números diferentes poderiam apresentar alta similaridade textual, mesmo quando seus valores eram significativamente distintos. Por exemplo, se a resposta correta fosse o número 1234 e o aluno respondesse 5234, a

comparação baseada em strings indicaria uma alta semelhança, apesar da diferença real de 4000. No corretor atual, a similaridade é calculada separadamente para números e strings, garantindo uma avaliação mais justa e eliminando esse erro. Esse processo é descrito em detalhes na sequência desta seção.

Outro problema que pode ocorrer é a necessidade de atribuir maior peso a certas palavras dentro da resposta. Considere, por exemplo, um exercício cuja resposta esperada seja definida pela classificação de um triângulo, permitindo três respostas de acordo com os lados do triângulo: (1) “De acordo com os lados, o triângulo é equilátero.”; (2) “De acordo com os lados, o triângulo é isósceles.”; (3) “De acordo com os lados, o triângulo é escaleno.”. Neste caso, as palavras “equilátero”, “isósceles” e “escaleno” são fundamentais, pois definem a resposta correta, enquanto o restante do texto é meramente contextual e idêntico para todas elas. Para lidar com essa situação, foi implementada uma função que calcula a similaridade levando em conta essas palavras-chave, denominadas palavras dinâmicas. A abordagem utilizada para essa análise é detalhada na sequência desta seção.

Para possibilitar as comparações diferenciadas entre *números* e *palavras dinâmicas*, foi necessário modificar o formato do arquivo de entrada (um exemplo é ilustrado na Figura 3.4), que agora segue a seguinte estrutura:

Figura 3.5 – Novo formato do arquivo de critérios.

```
3
1 1
Digite o número 1: ||entrada||0.5
2 1
Digite o número 2: ||entrada||0.5
3 4
A soma entre os números (...) e (...) é (...)||estático||1
10||número||1
20||número||1
30||número||6
```

Fonte: Próprio autor.

Conforme ilustrado na Figura 3.5, a primeira linha contém um número que indica a quantidade de linhas da resposta correta. As linhas seguintes seguem um padrão específico: cada uma contém dois números, sendo o primeiro a posição da linha atual e o segundo a quantidade de critérios dessa linha. A seguir, são listadas N linhas correspondentes aos critérios da linha atual, cada uma contendo três valores separados por “||”. Esses valores representam, respectivamente: (1) o conteúdo do critério; (2) seu tipo; (3) e sua classificação, que pode ser uma das seguintes:

- **entrada**: indica que a linha corresponde à leitura de dados;
- **estático**: representa uma linha contendo uma string fixa;
- **número**: indica que a linha contém um valor numérico;

- **dinâmico**: identifica uma palavra dinâmica dentro da resposta.

No caso de *strings* estáticas, o texto deve conter a parte fixa da linha e indicar os locais onde critérios numéricos ou dinâmicos devem ser inseridos, utilizando o símbolo "(...)". A associação entre esses símbolos e os critérios correspondentes ocorre por posição: o primeiro critério numérico ou dinâmico será inserido no primeiro "(...)", o segundo critério no segundo "(...)", e assim sucessivamente.

Embora essa abordagem represente uma grande melhoria na correção das questões, ela também apresenta algumas limitações. Um exemplo ocorre quando um número presente na resposta não deve ser tratado como tal pelo corretor. Considere o seguinte caso, em que a resposta esperada de um exercício, onde "X" corresponde ao valor resultante do processamento, é:

```
X pessoas precisam de vitamina B12.
```

Nesse contexto, o número 12 faz parte da parte estática da resposta, a respeito do nome da vitamina, e não deve ser separado para comparação numérica. No entanto, ao processar a resposta do aluno, o sistema irá identificar esse valor como um número e tratá-lo indevidamente. Atualmente, essa situação é evitada pela formulação dos enunciados, que não exigem respostas nesse formato. No entanto, essa limitação pode ser abordada em trabalhos futuros, explorando formas de distinguir corretamente números que fazem parte do texto fixo da resposta.

### 3.2.1 Diferenças entre a saída esperada e saída obtida

As diferenças destacadas na saída do programa, que identificavam os locais com divergências entre a saída obtida pelo código do aluno e a saída esperada, eram inicialmente marcadas utilizando *tags* HTML. Essas *tags* demarcavam o início e o fim das divergências entre as *strings* comparadas. No entanto, essa abordagem se mostrou inadequada, visto que a estilização e a utilização de *tags* HTML devem ser implementadas na camada de *front-end* do corretor automático. Para contornar essa limitação, foram realizadas modificações no código para que a marcação das diferenças fosse feita utilizando caracteres específicos.

As alterações foram implementadas na função `mark_differences`, a qual recebe duas *strings* como parâmetros e retorna essas *strings* com as divergências destacadas. O processo de destaque considera o tipo de diferença identificado entre as *strings*. Os tipos de diferenças possíveis são:

- *insert*: indica que algum caractere precisa ser adicionado para que a segunda *string* se torne idêntica à primeira;
- *replace*: indica que algum caractere da segunda *string* deve ser substituído para que coincida com o da primeira;

- *delete*: indica que algum caractere da segunda *string* deve ser removido para que ambas as *strings* sejam iguais.

Vale ressaltar que as marcações de diferenças são aplicadas em ambas as *strings*, visando tornar o mais claro possível para o aluno as divergências encontradas.

As diferenças que não penalizam na nota são aquelas em que o erro é apenas uma acentuação incorreta de uma letra, a presença ou ausência de pontuação (`!` ou `:`), ou espaços em branco em excesso ou em falta em qualquer uma das *strings* comparadas. Este tipo de erro é destacado de forma distinta para que a camada de *front-end* possa diferenciá-lo e evidenciá-lo para o aluno, por exemplo, exibindo-o com uma cor diferente dos erros que realmente penalizam na nota. Por outro lado, as diferenças que penalizam na nota incluem qualquer outro tipo de divergência, como a ausência ou excesso de caracteres na *string* gerada pelo código do aluno, ou a substituição de um caractere por outro incorreto.

Os marcadores utilizados são inseridos nas strings com a resposta correta e a resposta obtida pelo aluno, logo, esses marcadores foram escolhidos de forma a não serem confundidos com uma parte da string, ou seja, devem ser uma sequencia de caracteres improváveis de aparecer. Os marcadores escolhidos para cada situação foram:

- `@||@` para destacar os valores de entrada do programa.
- `@«@` para demarcar o início de uma sequência de caracteres que estão diferentes e penalizam na nota.
- `@»@` para demarcar o fim de uma sequência de caracteres que estão diferentes e penalizam na nota.
- `@^^@` para demarcar o início ou fim de uma sequência de caracteres que estão diferentes e não penalizam na nota.
- `@---@` usado na saída esperada nos casos em que era esperado uma linha em branco que não foi encontrada na resposta do aluno ou usado na saída obtida nos casos em que era esperado uma linha em branco que não foi encontrada na resposta do aluno.
- `@===@` usado na saída obtida nos casos em que era esperado uma linha em branco que não foi encontrada na resposta do aluno ou usado na saída esperada nos casos em que era esperado uma linha em branco que não foi encontrada na resposta do aluno. Também é utilizado em ambas saídas nos casos em que era esperado uma linha em branco e a mesma foi encontrada na resposta do aluno.

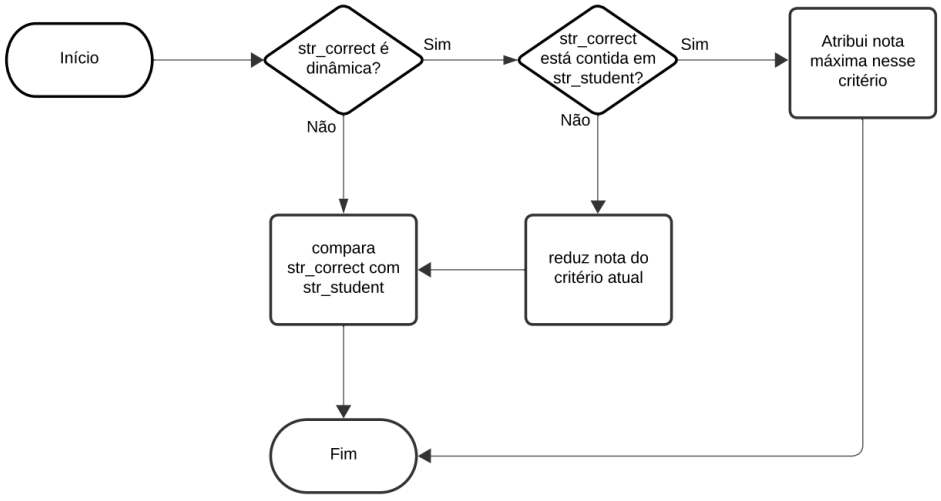
### 3.2.2 Similaridade entre textos

O cálculo da similaridade entre a saída esperada e a saída obtida pelo algoritmo do aluno era realizado utilizando a função `partial_ratio` da biblioteca *fuzz*. Nesta abordagem, cada critério da *string* correta era comparado com toda a saída gerada pelo código do aluno. Embora essa metodologia funcione adequadamente, ela trata todos os critérios uniformemente. No entanto, pode ser conveniente, em alguns casos, aplicar uma penalidade maior para erros que envolvam um critério que seja mais importante que os demais.

Retomemos o exemplo do problema em que, dado os lados de um triângulo, desejamos classificá-lo como "escaleno", "equilátero" ou "isósceles". Uma possível *string* de saída para esse exercício poderia ser: "`De acordo com os lados informados, o triângulo é equilátero.`". Nesse caso, a palavra "equilátero" possui um peso maior em relação ao restante da saída, pois o restante da frase é estático para todos os casos de teste, enquanto a palavra "equilátero" pode ser considerada dinâmica, já que poderia assumir outros valores, como "isósceles" ou "escaleno". Na nova função de comparação implementada, esse caso foi tratado de modo a permitir que o professor aplique uma penalidade maior quando o critério analisado for dinâmico.

A função desenvolvida recebe um novo parâmetro booleano para definir se o critério atual é dinâmico ou não, além de um parâmetro adicional (que deve estar entre 0 e 1) que representa a penalidade aplicada caso o critério seja dinâmico e esteja incorreto. A Figura 3.6 representa o fluxograma do novo algoritmo implementado.

Figura 3.6 – Fluxograma do algoritmo de comparação entre strings.




Fonte: Próprio autor.

Seja `str_correct` a *string* de critério avaliada no momento e `str_student` a saída gerada pelo código do estudante. Caso o critério não seja dinâmico, é feita apenas a comparação entre as *strings* utilizando a função `partial_ratio`, e a similaridade obtida define a nota para aquele critério. Vale ressaltar que, nos casos em que a similaridade é menor que 0.8, a nota atribuída é zero, conforme definido em Brito (2019). Nos casos em que o critério é dinâmico,

é verificado se `str_correct` está contida em `str_student`, ou seja, se há dentro da *string* `str_student` uma *string* exatamente igual a `str_correct`. Caso positivo, a nota máxima é atribuída; caso contrário, a nota será a similaridade de acordo com a função `partial_ratio`, multiplicada pela penalidade passada como parâmetro da função, sendo essa penalidade um valor entre 0 e 1.

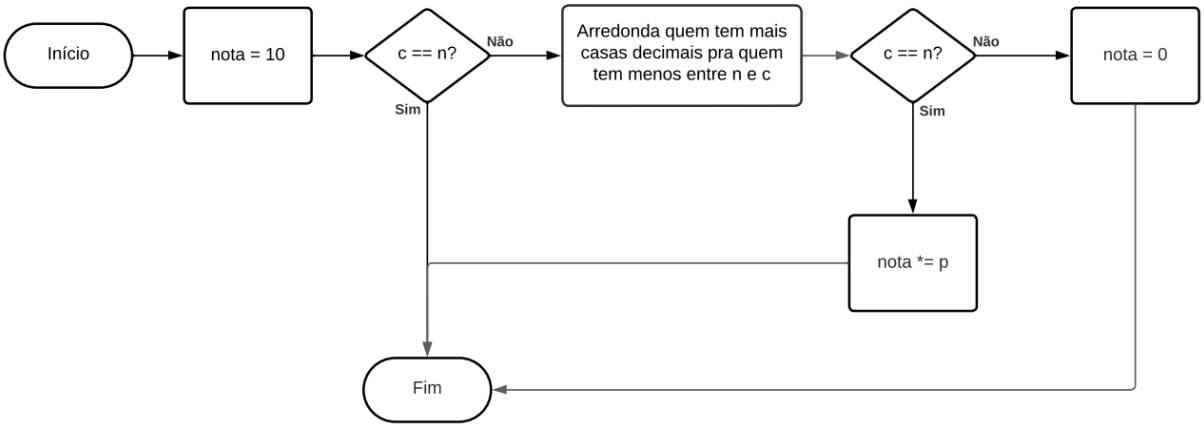
### 3.2.3 Similaridade entre Números

No processo de correção, inicialmente não se diferenciava se a comparação estava sendo feita entre números ou strings. A similaridade era calculada utilizando a função `partial_ratio`, que tratava números como se fossem strings. Para aprimorar esse processo e tornar a comparação mais precisa, foi necessário desenvolver uma função específica para calcular a similaridade entre o número correto e o número obtido como resposta pelo algoritmo do aluno.

Apenas verificar se os números são idênticos e atribuir a nota máxima em caso de igualdade, ou zero caso contrário, pode ser inadequado. Isso se aplica principalmente em exercícios que exigem uma resposta arredondada, onde o aluno pode ter realizado o cálculo correto, mas não ter arredondado o valor final ao imprimi-lo. Nesses casos, uma penalidade proporcional é aplicada, em vez de atribuir diretamente a nota zero.

A função desenvolvida recebe como parâmetros o número correto, o número obtido pelo aluno e um fator de penalidade (valor entre 0 e 1) que será aplicado nos casos em que o aluno acertou o número, mas errou na quantidade de casas decimais. A Figura 3.7 apresenta o fluxograma do algoritmo implementado.

Figura 3.7 – Fluxograma do algoritmo de comparação entre números.




Fonte: Próprio autor.

O algoritmo funciona da seguinte forma: inicialmente, considera-se a nota máxima. Seja $c$ o número correto e $n$ o número obtido pelo aluno. Se $c$ e $n$ forem iguais, a nota é mantida e o algoritmo termina. Caso contrário, o número com mais casas decimais é arredondado para a

quantidade de casas decimais do outro número. Se, após o arredondamento, os números forem iguais, aplica-se a penalidade e o algoritmo termina. Caso contrário, a nota é reduzida a zero.

Podem ocorrer duas situações em que a quantidade de casas decimais difere:

1. O aluno não arredondou a resposta, resultando em um número $n$ com mais casas decimais que o número correto $c$. Por exemplo, se $n = 10.8375$ e $c = 10.84$, o algoritmo irá arredondar $n$ para $10.84$, tornando-os iguais.

2. O aluno arredondou excessivamente, produzindo um número $n$ com menos casas decimais que o correto $c$. Por exemplo, se $n = 10.84$ e $c = 10.8375$, o algoritmo arredondará $c$ para $10.84$, igualando-os.

### 3.2.4 Método score_test_case

O método `score_test_case` engloba todas as operações necessárias para comparar o arquivo de saída do aluno com a resposta esperada, gerando, assim, o JSON contendo a nota e outras informações, que serão descritas na próxima subseção.

O pseudocódigo que representa o método score_test_case é mostrado a seguir:

**Algorithm 1** Calcula_nota_caso_teste

**Require:** Arquivo de critérios, Arquivo de saída do aluno, Arquivo de entradas do programa
**Ensure:** Dicionário contendo a avaliação da saída do aluno

```
Ler arquivos de critérios, saída do aluno e entradas do programa
linha_atual ← 0
total_linhas ← número total de linhas nos arquivos
while linha_atual < total_linhas do
    obter_informacoes_criterio()
    tratar_linhas_em_branco()
    obter_informacoes_saida_aluno()
    comparar_strings_estaticas()
    comparar_strings_dinamicas()
    comparar_numeros()
    construir_saida_esperada_e_obtida_com_marcacoes_de_diferencas()
    linha_atual ← linha_atual + 1
end while
dicionario_saida ← constroi_dicionario_saida()
return dicionario_saida
```

O método recebe como parâmetros os caminhos para o arquivo de critérios, o arquivo contendo a resposta do aluno e o arquivo com as entradas do programa. Inicialmente, realiza-se a leitura desses arquivos e, em seguida, inicia-se o loop principal do programa, que percorre linha por linha o arquivo de critérios. A cada iteração, são executadas, na seguinte ordem, as funções responsáveis pelo processamento e avaliação da saída do aluno:

#### 3.2.4.1 Função obter_informacoes_criterio

Essa função constrói uma lista contendo todos os critérios numéricos da linha atual, os critérios dinâmicos e, caso exista, o critério estático. Além disso, monta-se a *string* de saída completa, substituindo os símbolos "(...)" pelos respectivos valores, seguindo a ordem em que aparecem.

#### 3.2.4.2 Função tratar_linhas_em_branco

O tratamento de linhas em branco é realizado de forma que, para cada linha em branco no arquivo de critérios que não esteja no arquivo do aluno, adiciona-se o símbolo "`@===@`" na saída esperada e o símbolo "`@--@`" na saída obtida. Quando uma linha em branco é inserida pelo aluno indevidamente, adiciona-se o símbolo "`@--@`" na saída esperada e o símbolo "`@===@`" na saída obtida. Esse tratamento permite sinalizar na saída quando houver linhas em branco inesperadas, ou quando faltarem linhas em branco. Além disso, ele facilita a comparação entre as linhas que contêm conteúdo, evitando que a resposta do aluno seja considerada errada por conta da presença de linhas em branco. Vejamos o seguinte exemplo:

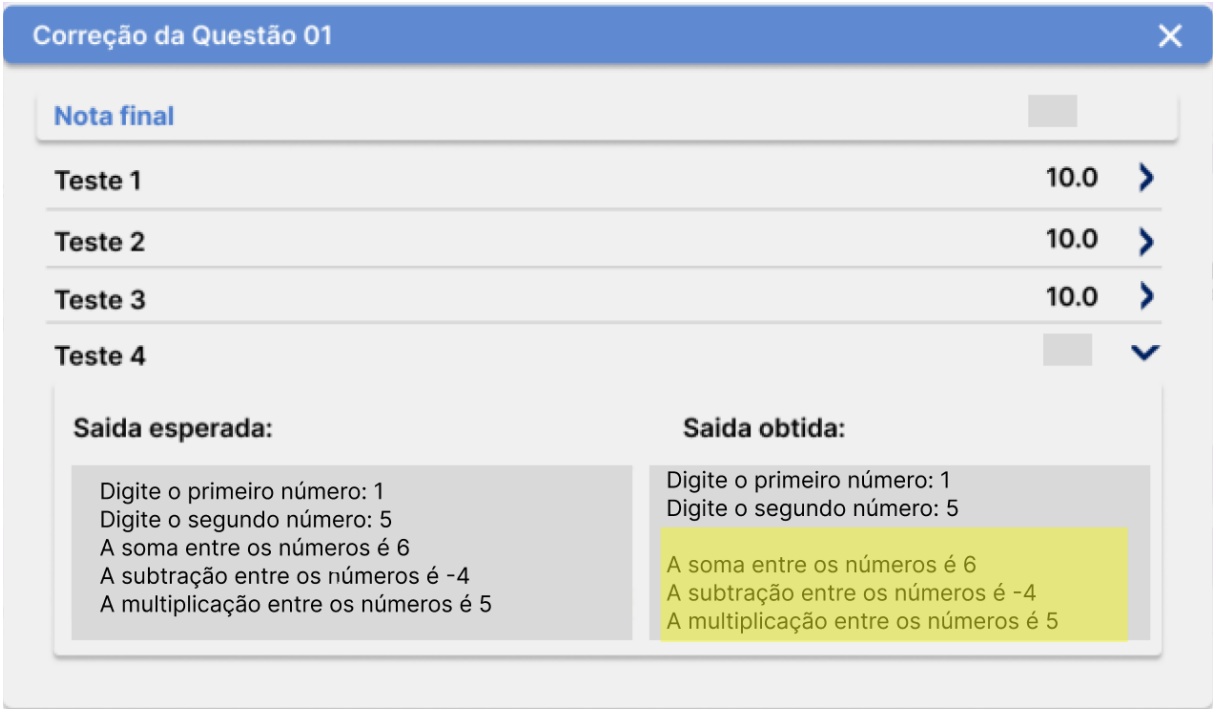
```
Saída esperada:

  1 Digite o primeiro número: 1
  2 Digite o segundo número: 5
  3 A soma entre os números é 6
  4 A subtração entre os números é -4
  5 A multiplicação entre os números é 5

Saída obtida:

  1 Digite o primeiro número: 1
  2 Digite o segundo número: 5
  3
  4 A soma entre os números é 6
  5 A subtração entre os números é -4
  6 A multiplicação entre os números é 5
```

Se a comparação fosse feita de forma síncrona, linha 1 com linha 1, linha 2 com linha 2, etc., o aluno receberia uma grande penalidade, pois, a partir da linha 3, a resposta correta do aluno estaria deslocada uma linha abaixo, resultando em uma penalização considerável. Com o tratamento aplicado, no entanto, essa linha em branco é simplesmente marcada com o caractere

especial, e a comparação continua da maneira correta. Assim, o aluno não sofre penalidade na nota, mas ainda pode visualizar o erro ao ter inserido uma linha a mais.

Antes das alterações, o corretor não penalizava indevidamente as linhas em branco, porém a marcação mostrada na interface do *opCodersJudge* ficava errada, como mostrado na Figura 3.8, marcando todas as linhas subsequentes à linha em branco, o que gerava dificuldade de interpretação do aluno.

Figura 3.8 – Exemplo da saída atual do corretor com linha em branco não esperada.




Fonte: Próprio autor.

#### 3.2.4.3 Função obter_informacoes_saida_aluno

Essa função constrói uma lista contendo todos os critérios numéricos da linha atual e uma *string* com as demais partes da linha concatenadas. Para separar os números dentro da *string*, utiliza-se a seguinte expressão regular: `(\d+\.\d+|\d+)`, que identifica tanto números inteiros quanto números reais.

Os algoritmos executados nas funções de comparação de strings estáticas, strings dinâmicas e a comparação numérica já foram descritos na Seção 4.2 e na Seção 3.2.3, respectivamente.

Na parte da comparação de *strings* dinâmicas, houve um problema gerado pelo novo formato dos critérios: a comparação da parte estática concatenada e a resposta do aluno nem sempre resulta na nota máxima, mesmo quando a resposta está correta. Tomemos como exemplo o seguinte caso:

**Exercício**: Escrever um programa que lê os números por extenso, calcula a soma e imprime a resposta por extenso.

- Resposta correta: “A soma entre os números dez e vinte é trinta”.
- Critérios concatenados: “A soma entre os números e é”.

Após o pré-processamento, que remove espaços e acentos, as *strings* comparadas serão:

- Resposta correta: `Asomaentreosnumerosdezevinteetrinta`
- Critérios concatenados: `Asomaentreosnumerosee`

A comparação entre essas duas *strings* não retorna 100% de similaridade quando feita pela função *partial_ratio*.

A solução adotada foi comparar as partes estáticas do critério separadamente e calcular a nota ponderando de acordo com o tamanho da *string*. Para o exemplo citado, a avaliação seria feita com as seguintes comparações:

1. `Asomaentreosnumeros` vs. `Asomaentreosnumerosdezevinteetrinta`
2. `e` vs. `Asomaentreosnumerosdezevinteetrinta`
3. `e` vs. `Asomaentreosnumerosdezevinteetrinta`

O total de caracteres do critério é 21, portanto, a comparação (1) tem peso $19/21$, enquanto as comparações (2) e (3) têm pesos $1/21$ cada.

### 3.2.5 Arquivo de saída JSON

O arquivo de saída gerado pela função era anteriormente composto pelas seguintes chaves: (i) `Caso`, (ii) `Erro`, (iii) `Pontuacao`, (iv) `SaidaEsperada`, e (v) `SaidaObtida`. Um exemplo do JSON gerado pode ser visto na Figura 3.9.

Figura 3.9 – Exemplo do novo JSON gerado.


```
{
    Caso : 0
    Erro : false
    Pontuacao : 9.09
    SaidaEsperada : Digite o número 1: @||@dez@||@
                    Digite o número 2: @||@vinte@||@
                    A soma @<<@entre @>>@os @<<@n@>>@úmeros dez e vinte é trinta

    SaidaObtida : Digite o número 1: @||@dez@||@
                  Digite o número 2: @||@vinte@||@
                  A soma @<<@d@>>@os @<<@m@>>@úmeros dez e vinte é trinta

    Criterios : [ 3 items ]
}
```


Fonte: Próprio autor.

O novo formato do arquivo inclui uma chave adicional, `Criterios`, que armazena uma lista onde cada elemento representa uma linha de critérios. Cada um desses elementos é, por sua vez, uma lista contendo a quantidade de critérios daquela linha. Cada critério é representado por uma tripla (armazenada como uma lista), contendo: o conteúdo do critério, o valor atribuído a ele e a nota obtida. A Figura 3.10 ilustra um exemplo do conteúdo da chave `Criterios` dentro do JSON gerado como resposta.

Figura 3.10 – Exemplo da chave `Criterios` do novo JSON gerado.


```
Criterios : [ 3 items
  0 : [ 1 item
    0 : [ 3 items
        0 : Digite o número 1:
        1 : 0.5
        2 : 0.5
    ]
  ]
  1 : [ 1 item
    0 : [ 3 items
        0 : Digite o número 2:
        1 : 0.5
        2 : 0.5
    ]
  ]
  2 : [ 4 items
    0 : [ 3 items
        0 : A soma entre os números (...) e (...) é (...)
        1 : 1.0
        2 : 0.09523809523809523
    ]
    1 : [ 3 items
        0 : dez
        1 : 1.0
        2 : 1.0
    ]
    2 : [ 3 items
        0 : vinte
        1 : 1.0
        2 : 1.0
    ]
    3 : [ 3 items
        0 : trinta
        1 : 6.0
        2 : 6.0
    ]
  ]
]
}
```


Fonte: Próprio autor.

Essa nova chave é essencial para a implementação, no *front-end*, de uma funcionalidade que permite ao aluno visualizar o valor obtido em cada critério do exercício, além da nota total. Isso facilita a identificação de erros e proporciona um melhor entendimento sobre a avaliação final.

O código e testes implementados estão disponíveis no *Github* [2].

[2] Código: <https://github.com/Leandro-Rodrigues/opcoders-automatic-corrector/tree/develop/TesteCorretor/newCorrector>

# 4 Resultados

Neste capítulo, apresentam-se os resultados obtidos a partir da implementação do método que computa o resultado do corretor automático, analisando seu desempenho em diferentes cenários. Inicialmente, são descritos os testes realizados para validar a precisão do sistema, comparando as saídas esperadas e as saídas geradas para diferentes tipos de respostas dos alunos.

Além disso, avalia-se a eficácia das melhorias introduzidas, destacando como o novo formato de critérios impacta a correção e a interpretação das notas. Por fim, discutem-se as limitações do modelo atual e possíveis aprimoramentos futuros.

Na Seção 4.1, são apresentados os resultados dos testes unitários realizados no método responsável por evidenciar as diferenças entre as saídas esperadas e obtidas. Já na Seção 4.2, são exibidos os resultados dos testes unitários do método que calcula a similaridade entre textos. Da mesma forma, a Seção 4.3 apresenta os resultados dos testes unitários aplicados ao método de cálculo da similaridade entre números. Por fim, na Seção 4.4, é exibida uma projeção da nova interface do corretor, considerando as melhorias implementadas no *back-end* ao longo deste trabalho.

## 4.1 Diferenças entre a saída esperada e saída obtida

Foram realizados oito testes unitários utilizando a biblioteca `pytest` para assegurar que a função `mark_differences` apresenta o comportamento esperado. Os testes abrangeram os possíveis casos e verificaram se a resposta obtida pela função corresponde à resposta esperada.

### 4.1.1 Testes com diferenças que não penalizam na nota

Como demonstrado nos testes da Tabela 4.1, a nova convenção emprega os caracteres @^^@ para demarcar diferenças que não penalizam na nota, ou seja, os caracteres @^^@ demarcam o início e o fim da parte diferente entre as strings.

Tabela 4.1 – Resultado dos testes com diferenças que não penalizam na nota.

| **Resposta correta** | **Resposta do aluno** | **String correta com diferenças destacadas** | **String do aluno com diferenças destacadas** |
|---|---|---|---|
| “saida esperada” | “saida esperada” | “saida esperada” | “saida esperada” |
| “aa aa” | “aa  aa” | “aa aa” | “aa @^^@  @^^@ aa” |
| “abçcd” | “abccd” | “ab@^^@ç@^^@cd” | “ab@^^@c@^^@cd” |
| “abãcd” | “abacd” | “ab@^^@ã@^^@cd” | “ab@^^@a@^^@cd” |
| “abácd” | “abacd” | “ab@^^@á@^^@cd” | “ab@^^@a@^^@cd” |
| “abõcd” | “abocd” | “ab@^^@õ@^^@cd” | “ab@^^@o@^^@cd” |
| “abécd” | “abecd” | “ab@^^@é@^^@cd” | “ab@^^@e@^^@cd” |

Fonte: Próprio autor.

### 4.1.2 Testes com diferenças que penalizam na nota

Como demonstrado nos testes da Tabela 4.2, a nova convenção emprega os caracteres @<<@ e @>>@ para marcar as diferenças que penalizam na nota.

Tabela 4.2 – Resultado dos testes com diferenças que penalizam na nota.

| | |
|---|---|
| **Resposta correta** | "Saída correta esperada = 5" |
| **Resposta do aluno** | "Xaida que corretá esperada é 9" |
| **String correta com diferenças destacadas** | "@«@S@»@a@^^@ í@^^@ da corret@^^@ a@^^@ esperada @«@= 5@»@" |
| **String do aluno com diferenças destacadas** | "@«@X@»@a@^^@i@^^@da@^^@ @^^@ @«@que@»@ corret@^^@á@^^@ esperada @«@é 9@»@" |

Fonte: Próprio autor.

## 4.2 Similaridade entre textos

Foram realizados cinco testes unitários na função de comparação de strings, visando cobrir diferentes casos e verificar se o seu comportamento está correto para todas as possibilidades.

### 4.2.1 Testes sem palavras dinâmicas

A Tabela 4.3 ilustra três exemplos das strings fornecidas pelo programa de um estudante, as respostas esperadas e a similaridade entre ambas. É importante destacar que, para os casos de comparação de *strings* estáticas, ou seja, aquelas que não contêm palavras dinâmicas, apenas o valor da similaridade retornado pela função `partial_ratio` foi considerado.

Tabela 4.3 – Resultado dos testes variando a resposta do aluno.

| **Resposta correta** | **Resposta do aluno** | **Similaridade(%) / Nota** |
|---|---|---|
| "resposta correta" | "resposta errada" | 73 / 0.00 |
| "resposta correta" | "resposta certaaa" | 81 / 0.81 |
| "resposta correta" | "resposta correta" | 100 / 1.00 |

Fonte: Próprio autor.

No primeiro teste, que compara "resposta correta" com "resposta errada", a função `partial_ratio` retorna uma similaridade de 73%. Como há a restrição de que similaridades inferiores a 80% são consideradas *strings* muito diferentes, a nota obtida é zero.

No segundo teste, que compara "resposta correta" com "resposta corretaaa", a função `partial_ratio` retorna uma similaridade de 81%, resultando em uma nota de 0.81.

No último teste, ambas as *strings* são idênticas, ou seja, a função `partial_ratio` retorna uma similaridade de 100%, garantindo a nota máxima.

### 4.2.2 Testes com palavras dinâmicas

Nos testes desta seção, considerou-se o problema de classificar um triângulo de acordo com seus lados, sendo a resposta esperada a seguinte: "`O triângulo calculado é equilátero de acordo com os lados.`". A avaliação foi dividida em três partes:

- Parte 1: "`O triângulo calculado é`", com peso de $20/39 * 0.3$ na nota final, sendo 20 o número de caracteres na parte do texto que será comparada, 39 é o número de caracteres no texto todo e 0.3 o peso desse critério. Vale ressaltar que o caractere de espaço não é contabilizado.
- Parte 2: "`equilátero`", com peso de 0.7 na nota final, sendo este um critério no qual a palavra é dinâmica.
- Parte 3: "`de acordo com os lados`", com peso de $19/39 * 0.3$ na nota final, sendo 19 o número de caracteres na parte do texto que será comparada, 39 é o número de caracteres no texto todo e 0.3 o peso desse critério. Vale ressaltar que o caractere de espaço não é contabilizado.

Cada parte da *string* foi comparada com toda a *string* gerada pelo código do aluno. Vamos supor que a *string* `str_student` seja: "`O triângulo cauculado é equilatero de acordo com a área.`", e que o parâmetro $p$ seja 0.5. A Tabela 4.4 apresenta os resultados de similaridade para o exemplo citado.

#### 4.2.2.1 Testes com a palavra dinâmica correta

Tabela 4.4 – Resultado dos testes com a palavra dinâmica correta.

| Resposta correta | Resposta do aluno | Similaridade(%) / Nota |
|---|---|---|
| "O triângulo calculado é " | "O triângulo cauculado é equilatero de acordo com a área." | 95 / 0.95 |
| "equilátero" | "O triângulo cauculado é equilatero de acordo com a área." | 100 / 1.00 |
| " de acordo com os lados" | "O triângulo cauculado é equilatero de acordo com a área." | 69 / 0.00 |

Fonte: Próprio autor.

A similaridade da terceira parte foi de 69%. No entanto, como esse valor é inferior a 80%, a nota atribuída a essa parte é 0. Com base nesses resultados, a nota final é 8.46, obtida pela soma de dois valores:

- O primeiro termo, 7, corresponde à nota máxima atribuída ao critério dinâmico.
- O segundo termo, $\frac{20}{39} \times 0.3 \times 0.95 \times 10$, é composto por: $\frac{20}{39}$, que representa o peso dessa parte da *string*; $0.3$, que corresponde ao peso do critério; $0.95 \times 10$, que indica a similaridade ajustada ao intervalo de 0 a 10.

#### 4.2.2.2 Teste com a palavra dinâmica errada

Para o teste com a palavra dinâmica errada, considerou-se um parâmetro de penalização $p = 0.9$ para avaliar a precisão da resposta gerada pelo código do aluno em relação à palavra dinâmica incorreta. A tabela abaixo mostra os resultados da similaridade para cada critério. Suponha que a string `str_student` seja: "`O triangulo cauculado é equalatero de acordo com a área.`". A Tabela 4.5 apresenta os resultados de similaridade para o exemplo citado.

Tabela 4.5 – Resultado dos testes com a palavra dinâmica errada.

| **Resposta correta** | **Resposta do aluno** | **Similaridade(%) / Nota** |
|---|---|---|
| “O triângulo calculado é ” | “O triangulo cauculado é equalatero de acordo com a área.” | 95 / 0.95 |
| “equilátero” | “O triangulo cauculado é equalatero de acordo com a área.” | 90 / 0.81 |
| “ de acordo com os lados” | “O triangulo cauculado é equalatero de acordo com a área.” | 69 / 0.00 |

Fonte: Próprio autor.

A similaridade da terceira parte foi de 69%. No entanto, como esse valor é inferior a 80%, a nota atribuída a essa parte é 0. Com base nos resultados obtidos, a nota final é 6.16, calculada a partir da soma de dois valores:

- O primeiro termo, 5.67, é obtido por $0.7 \times 0.9 \times 0.9 \times 10$, onde: $0.7$ representa o peso do critério; $0.9$ é o parâmetro de penalidade; $0.9 \times 10$ corresponde à similaridade escalada para o intervalo de 0 a 10.
- O segundo termo, $\frac{20}{39} \times 0.3 \times 0.95 \times 10$, é composto por: $\frac{20}{39}$, que representa o peso dessa parte da *string*; $0.3$, que corresponde ao peso do critério; $0.95 \times 10$, que indica a similaridade ajustada ao intervalo de 0 a 10.

## 4.3 Similaridade entre Números

Foram realizados sete testes unitários na função de comparação de números, com o objetivo de cobrir diferentes casos e verificar se o seu comportamento está correto para todas as possibilidades.

### 4.3.1 Testes com Números Inteiros

Nos testes realizados com números inteiros, os resultados foram os apresentados na Tabela 4.6.

Tabela 4.6 – Resultado dos testes com números inteiros.

| Número correto | Número do aluno | Similaridade (%) |
|---|---|---|
| 1 | 1 | 100 |
| 1 | 2 | 0 |

Fonte: Próprio autor.

Nestes casos, o comportamento é direto: se os números são iguais, a similaridade é de 100%, caso contrário, a similaridade é 0%.

### 4.3.2 Testes com Números Decimais

Nos testes com números decimais, foi definido um valor de 0,5 ($p = 0{,}5$) para representar a penalidade aplicada na nota do aluno nos casos em que apenas o número de casas decimais está incorreto. Os resultados obtidos foram os apresentados na Tabela 4.7.

Tabela 4.7 – Resultado dos testes com números decimais.

| Caso | Número correto | Número do aluno | Similaridade (%) |
|---|---|---|---|
| 1 | 1,6767 | 1,6767 | 100 |
| 2 | 1,67 | 1,6767 | 0 |
| 3 | 1,68 | 1,6767 | 50 |
| 4 | 1,6767 | 1,67 | 0 |
| 5 | 1,6767 | 1,68 | 50 |

Fonte: Próprio autor.

Os resultados indicam que:

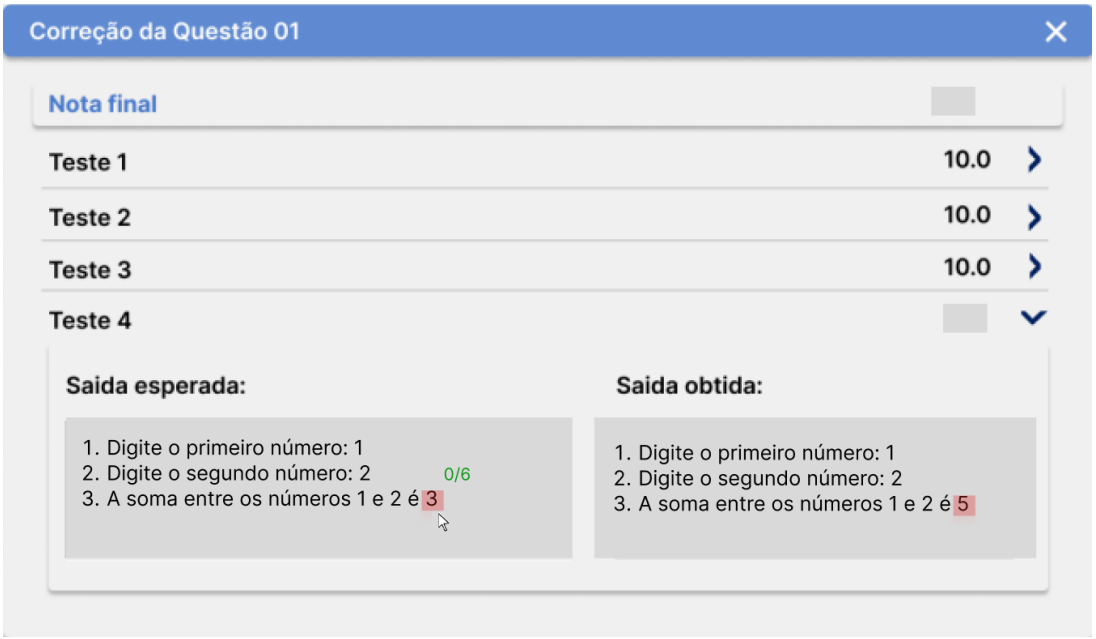
- Números exatamente iguais recebem a nota máxima, ou seja, 100% de similaridade (caso 1).
- Números iguais, mas com diferentes quantidades de casas decimais, recebem metade da nota, considerando que $p = 0{,}5$ (casos 3 e 5).
- Números que permanecem diferentes mesmo após o arredondamento das casas decimais recebem nota zero (casos 2 e 4).

## 4.4 Projeção de Resultado Final

Com a introdução das novas marcações para diferenciar erros que penalizam e os que não impactam na nota, além da exibição das notas separadas para cada critério, o *front-end* poderá apresentar uma tela detalhada com todas as informações da correção do exercício. Isso permitirá que o aluno compreenda melhor a nota obtida. Um possível *design* dessa nova tela pode ser observado na Figura 4.1, na Figura 4.2 e na Figura 4.3.

Na Figura 4.1, observa-se um caso em que o aluno cometeu um erro no resultado da soma solicitada pelo exercício. Como esse erro impacta diretamente na nota, a correção é destacada com uma marcação vermelha. Além disso, ao posicionar o cursor sobre a parte marcada, a nota correspondente a esse critério é exibida juntamente com seu valor total.

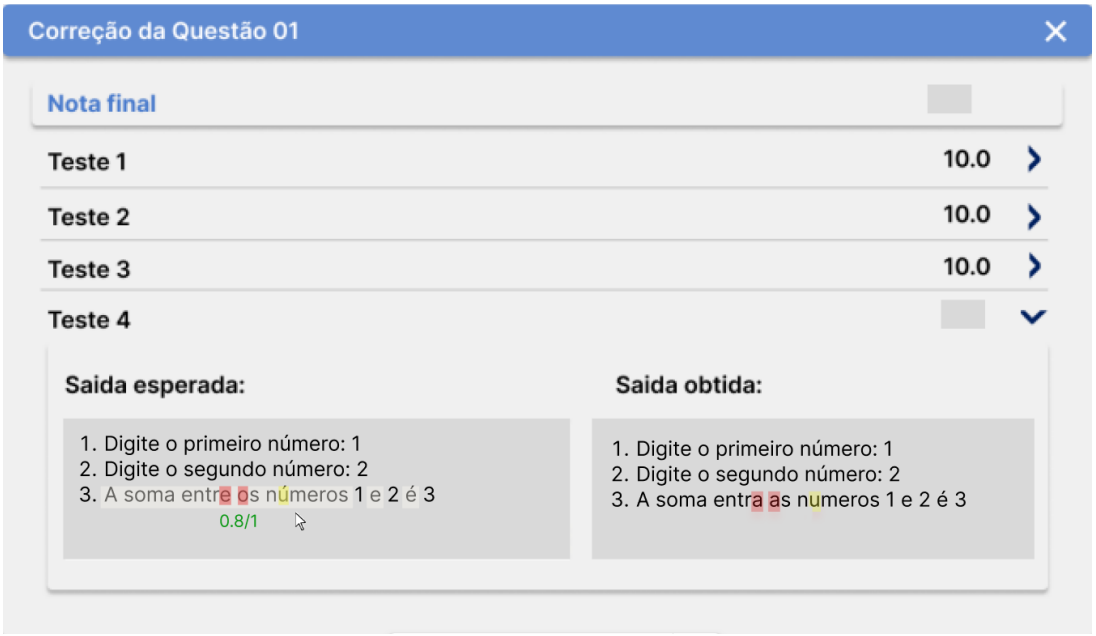
Figura 4.1 – Exemplo de marcação para um erro que penaliza a nota.




Fonte: Próprio autor.

Já na Figura 4.2, são ilustrados dois tipos de diferenças: aquelas que penalizam a nota e aquelas que não afetam a pontuação. Como os marcadores enviados pelo *back-end* distinguem esses casos, o *front-end* pode representar essa distinção visualmente. No exemplo apresentado, diferenças que resultam em penalização são destacadas em vermelho, enquanto diferenças que não afetam a nota são exibidas em amarelo.

Figura 4.2 – Exemplo de marcações para erros que penalizam e que não penalizam.

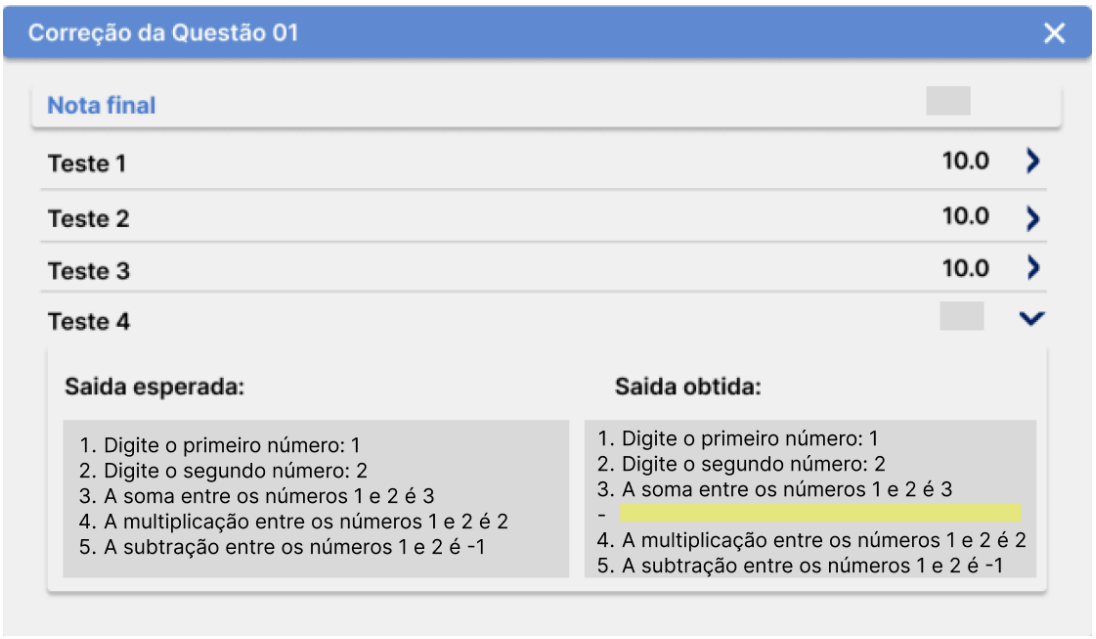



Fonte: Próprio autor.

Além disso, ao posicionar o cursor sobre um trecho do texto, todo o critério correspondente

é realçado em branco. Isso permite que o aluno identifique exatamente quais partes do texto foram consideradas na avaliação do critério, cuja nota também é exibida junto ao seu valor total.

Figura 4.3 – Exemplo de marcações para linha em branco errada.




Fonte: Próprio autor.

Na Figura 4.3, observa-se a melhoria no tratamento de linhas em branco indevidas no código do aluno. Enquanto na versão anterior todas as linhas subsequentes poderiam ser afetadas, nesta nova abordagem apenas a linha incorreta é destacada. Essa alteração torna a correção mais precisa e facilita a compreensão do erro pelo aluno.

## 4.5 Impacto esperado

Com os resultados obtidos nos testes unitários realizados nos métodos de marcação de diferenças, similaridade entre textos, similaridade entre números e na correção geral, foi possível validar as melhorias implementadas no algoritmo. A nova abordagem para comparação de *strings* dinâmicas demonstrou um aumento na precisão da avaliação, tornando a nota atribuída mais justa ao levar em consideração as variações esperadas nas respostas. Da mesma forma, os testes no método de comparação de números evidenciaram que a separação de valores numéricos possibilita uma avaliação mais coerente, reduzindo penalizações indevidas em situações onde pequenos desvios podem ocorrer. Já na correção geral, os resultados confirmam que a nova estrutura permite uma análise mais detalhada e modularizada das respostas, garantindo maior flexibilidade para futuras melhorias.

Com essas otimizações, espera-se que os alunos tenham uma experiência mais clara e intuitiva ao interpretar os resultados de suas submissões. A distinção visual entre erros que impactam a nota e aqueles que não penalizam permite que o estudante compreenda melhor

quais aspectos de sua resposta precisam ser ajustados. Além disso, a exibição da nota separada por critérios possibilita uma análise mais detalhada do desempenho, tornando o processo de aprendizado mais direcionado e eficiente.

Outro impacto significativo está na transparência da correção. Com a nova abordagem, os alunos conseguem visualizar exatamente quais partes de sua resposta foram analisadas e como cada critério contribuiu para a nota final. Essa clareza reduz possíveis dúvidas sobre o processo avaliativo e proporciona um *feedback* mais objetivo, auxiliando na escrita de programas corretos e bem estruturados.

# 5 Considerações Finais

Este capítulo apresenta as considerações finais sobre o trabalho realizado. Na Seção 5.1, são discutidas as principais conclusões obtidas, considerando um panorama geral das contribuições desenvolvidas. Em seguida, na Seção 5.2, são exploradas possíveis melhorias e extensões que podem ser implementadas para aprimorar ainda mais o corretor.

## 5.1 Conclusão

A parte do *back-end* do corretor *opCodersJudge* responsável pela correção de códigos estava desatualizada. Para os desenvolvedores, a estrutura e a legibilidade do código dificultavam a implementação de novas funcionalidades. Para o usuário final, o aluno, a correção apresentava inconsistências – por exemplo, a marcação inadequada de linhas em branco – que comprometiam a clareza dos resultados. Essa defasagem motivou a necessidade de uma série de melhorias, as quais foram implementadas e detalhadas neste trabalho.

A refatoração permitiu a criação de um código mais limpo, organizado e aderente às boas práticas da linguagem Python, facilitando a manutenção e a expansão do sistema. Com essas alterações, futuros programadores encontrarão um ambiente mais propício para o desenvolvimento de novas funcionalidades no *back-end* do corretor.

Os resultados obtidos com o novo algoritmo de correção demonstraram que a separação dos números para uma avaliação personalizada torna a nota final mais justa e coerente em comparação com o modelo anterior. Da mesma forma, a análise específica de *strings* dinâmicas, parte essencial da resposta do exercício, passou a ter uma abordagem mais criteriosa, gerando uma resposta mais justa.

Além disso, houve um ganho significativo na visualização das saídas esperadas e obtidas. A utilização de diferentes caracteres para demarcar as diferenças que penalizam e as que não penalizam na nota possibilita a criação de uma interface intuitiva, capaz de destacar, de forma clara, onde ocorrem as discrepâncias. Essa nova saída permite que a interface apresente, de maneira detalhada, as notas atribuídas a cada critério, contribuindo para uma melhor compreensão da avaliação pelo aluno. Também foi corrigido o problema relacionado às linhas em branco, aprimorando a exibição dos resultados.

Em suma, as melhorias implementadas promovem um aprendizado mais rápido e eficiente, uma vez que os erros nos programas dos alunos são evidenciados de forma clara e precisa, auxiliando na identificação e correção dos problemas.

## 5.2 Trabalhos Futuros

Diante das significativas melhorias implementadas no *back-end* do corretor, a próxima etapa essencial é a adaptação do *front-end* para processar e exibir corretamente a nova saída gerada. Essa integração é fundamental para proporcionar uma experiência mais clara e intuitiva ao usuário final, garantindo que os benefícios do aprimoramento do algoritmo de correção sejam plenamente aproveitados. Também será necessária uma reestruturação dos arquivos de critérios dos exercícios para seguirem o novo formato proposto neste trabalho.

Além dessas adaptações, ao longo do desenvolvimento deste trabalho, foram identificadas outras oportunidades de melhoria, que podem ser exploradas em pesquisas futuras:

- **Aprimoramento da análise estática**: A análise estática pode ser expandida para tornar o sistema ainda mais completo, identificando padrões mais sofisticados no código do aluno e proporcionando uma correção mais precisa.
- **Refinamento na comparação de números e *strings* dinâmicas**: Diversas abordagens foram levantadas para tratar esses elementos da melhor forma. No entanto, um estudo mais aprofundado pode permitir melhorias no processo, ampliando sua abrangência para lidar com casos específicos ainda não contemplados, como números presentes na resposta esperada que fazem parte da estrutura textual e não devem ser considerados como valores numéricos na correção. Uma abordagem que permita diferenciar esses casos pode tornar a correção ainda mais precisa.
- **Melhoria na correspondência entre as linhas da solução esperada e a resposta do aluno**: A correspondência entre essas linhas pode ser otimizada para garantir uma avaliação mais justa, reduzindo inconsistências na comparação e proporcionando uma visualização mais clara das diferenças.

A implementação dessas melhorias contribuirá para um corretor mais preciso, flexível e eficiente, consolidando sua utilidade tanto para alunos quanto para professores.

# Referências


BALL, T. The concept of dynamic analysis. ACM SIGSOFT Software Engineering Notes, ACM New York, NY, USA, v. 24, n. 6, p. 216–234, 1999.

BEECROWD. lista de submissões no Beecrowd. 2024. Disponível em: <https://www.beecrowd.com.br/judge/pt>.

BRITO, P. S.; FORTES, R. S. O uso de corretores automáticos para o ensino de programaç ao de computadores para alunos de engenharia. 2019.

BRITO, P. S. S. O uso de ferramentas computacionais para o ensino de programação para alunos de engenharia. UFOP, 2019. Disponível em: <http://www.monografias.ufop.br/handle/35400000/2274>.

BROOKE, J. Sus: a "quick and dirty'usability. Usability evaluation in industry, Taylor & Francis, v. 189, n. 3, p. 189–194, 1996.

CEDRAZ, V. F. Uma avaliação de usabilidade do corretor de exercícios de introdução à programação opcoders judge. UFOP, 2023. Disponível em: <http://www.monografias.ufop.br/handle/35400000/5507>.

CRUZ, A. K. B. S. da; NETO, C. d. S. S.; CRUZ, P. T. M. B. da; TEIXEIRA, M. A. M. Utilizaçao da plataforma beecrowd de maratona de programaçao como estratégia para o ensino de algoritmos. In: SBC. Anais Estendidos do XXI Simpósio Brasileiro de Jogos e Entretenimento Digital. [S.l.], 2022. p. 754–764.

LANG, P. Behavioral treatment and bio-behavioral assessment: Computer applications. Technology in mental health care delivery systems, Ablex, p. 119–137, 1980.

LISBACH, B.; MEYER, V. Linguistic identity matching. [S.l.]: Springer, 2013.

LOURIDAS, P. Static code analysis. Ieee Software, IEEE, v. 23, n. 4, p. 58–61, 2006.

MARCUSSI, L. D.; GUEDES, K.; FILHO, R. G. D. M.; FILHO, R. M. S.; JUNIOR, C. R. B. Pesquisa no ensino de algoritmos e programação nas engenharias: Estudos e resultados preliminares. In: Simpósio de Engenharia de Produção. [S.l.: s.n.], 2016.

MENDONçA, T. S. Projeto e desenvolvimento de uma plataforma de gestão de questões dinâmicas para o ensino de programação de computadores. UFOP, 2023. Disponível em: <http://www.monografias.ufop.br/handle/35400000/5996>.

NAZÁRIO, D. C.; SOUZA, A. de. Boca-lab: Corretor automático de código adaptado ao ensino de linguagem de programação. Anais do Computer on the Beach, p. 42–46, 2010.

NIELSEN, J. Usability inspection methods. In: Conference companion on Human factors in computing systems. [S.l.: s.n.], 1994. p. 413–414.

NTNU, L. M. MC646 - Aula 03 Parte 1/2 - Análise Estática de Código. 2020. Acessado em fevereiro de 2024. Disponível em: <https://www.youtube.com/watch?v=W1tr1j-_TMY>.

PATROCíNIO, J. A. d. Opcoders judge : uma versão online para o corretor automático de exercícios de programação do projeto opcoders. UFOP, 2023. Disponível em: <http://www.monografias.ufop.br/handle/35400000/5360>.

STRIEWE, M.; GOEDICKE, M. A review of static analysis approaches for programming exercises. In: SPRINGER. International Computer Assisted Assessment Conference. [S.l.], 2014. p. 100–113.