

Juiz de Código Adaptativo: Uma Abordagem para Avaliação Justa de Algoritmos entre C++ e Python no Contexto Educacional

Vandressa Galdino Soares¹, Wilkerson L. Andrade¹

¹Unidade Acadêmica de Sistemas e Computação – Universidade Federal de Campina Grande (UFCG) – Campina Grande – PB – Brasil

vandressa.soares@ccc.ufcg.edu.br, wilkerson@computacao.ufcg.edu.br

Abstract. *Online code judges, widely used in programming education, generally adopt fixed time limits calibrated for C++. This model penalizes Python solutions of adequate complexity with Time Limit Exceeded (TLE) verdicts that reflect execution overhead rather than algorithmic quality. This work aims to promote fairness across languages by proposing and validating an Adaptive Code Judge, which calibrates time limits per language with benchmarks in Docker containers over synthetic classes and eleven CSES problems, with verdicts from the official judge. The adaptive limit eliminates unjust TLE without admitting incorrect or inefficient solutions, with factors from $3\times$ to $120\times$ depending on the problem.*

Resumo. *Juízes de código online, amplamente usados no ensino de programação, adotam em geral tempos-limite fixos calibrados em C++. Esse modelo penaliza soluções Python de complexidade adequada com vereditos de Time Limit Exceeded (TLE) que refletem o overhead de execução, não a qualidade algorítmica. Este trabalho tem como objetivo promover equidade entre linguagens, propondo e validando um Juiz de Código Adaptativo, que calibra os tempos-limite por linguagem com benchmarks em contêineres Docker sobre classes sintéticas e onze problemas do CSES, com vereditos do juiz oficial. O limite adaptativo elimina o TLE injusto sem admitir soluções incorretas nem ineficientes, com fatores de $3\times$ a $120\times$ conforme o problema.*

Palavras-chave: *Juiz Online; Avaliação Automática de Código; Tempo-Limite Adaptativo; Docker; Equidade de Linguagens; Benchmarking Estatístico.*

1. Introdução

A avaliação automatizada de código é amplamente utilizada no ensino de Ciência da Computação [Ihantola et al. 2010, Silva et al. 2024], permitindo feedback imediato aos estudantes e auxiliando docentes no acompanhamento pedagógico. Para viabilizar esse processo, recorre-se amplamente aos juízes online, que se consolidaram como referência por disponibilizarem vastos repositórios de problemas e contarem com validação contínua por milhares de usuários. Esses sistemas, entretanto, estão fortemente inseridos no contexto da programação competitiva — em plataformas como Codeforces, CSES, UVA, AtCoder e USACO — e adotam um modelo de correção baseado em tempos-limite fixos definidos a partir do desempenho de soluções em C++ [Wasik et al. 2018].

Esse paradigma, quando aplicado ao ensino, gera distorções. Linguagens interpretadas, como Python, frequentemente recebem vereditos de Time Limit Exceeded (TLE) mesmo em soluções de complexidade adequada, por diferenças inerentes de execução, como o overhead de interpretação e limites de pilha mais restritos. Na prática, esse cenário restringe tanto o conjunto de problemas aplicáveis quanto as linguagens viáveis na disciplina, e desvia o foco da avaliação da técnica algorítmica para o desempenho do modelo de execução. Além disso, tende a comprometer a motivação dos estudantes, pois episódios repetidos de insucesso em juízes online estão associados a frustração e à redução da efetividade do aprendizado [Wang et al. 2023].

Diante desses pontos, evidencia-se a necessidade de mecanismos que promovam equidade entre linguagens no contexto educacional, possibilitando avaliações mais justas, centradas na correção e na complexidade assintótica da solução. A aplicação da lógica competitiva, embora adequada em competições de programação, mostra-se inadequada quando transferida para o ensino superior de Computação, em especial para disciplinas de algoritmos e estruturas de dados da graduação, como Análise de Técnicas de Algoritmos, nas quais tópicos mais avançados — como grafos, programação dinâmica e recursão profunda — colocam em desvantagem estudantes que utilizam linguagens mais didáticas. É nesse nível de ensino, em que o tempo-limite atua como critério de avaliação, que a ferramenta proposta se insere.

Este trabalho investiga se é possível promover equidade entre linguagens em juízes online por meio de calibração baseada em benchmarks controlados. Para isso, propõe e avalia o Juiz de Código Adaptativo, que calibra os tempos-limite por linguagem a partir de medições empíricas entre C++ e Python. Escolheu-se esse par por representar o contraste central do problema: C++, linguagem de alto desempenho e referência das plataformas, e Python, interpretada e amplamente adotada no ensino introdutório de programação [Siegfried et al. 2021]. A contribuição é investigar empiricamente se o modelo dinâmico reduz penalizações injustas, amplia o repertório de problemas aplicáveis e oferece avaliações mais alinhadas ao propósito pedagógico.

Nessas questões, multiplicadores fixos são fatores constantes (usualmente $2\times$ ou $3\times$) aplicados por algumas plataformas sobre o tempo-limite do C++, e o fator de calibração β é a razão empírica entre os tempos de execução de Python e C++ medidos para cada problema (Seção 3.3). Posto isso, este trabalho define três questões de pesquisa:

- **QP1.** Os multiplicadores de tempo fixos ($2\times$ ou $3\times$ sobre C++), adotados por juízes online tradicionais, capturam adequadamente as diferenças de execução entre C++ e Python ao longo de diferentes classes de complexidade algorítmica?
- **QP2.** A calibração adaptativa de tempos-limite, baseada em fatores empíricos (β), reduz a ocorrência de vereditos Time Limit Exceeded injustos em soluções Python corretas, sem comprometer a seletividade contra soluções ineficientes?
- **QP3.** O fator de calibração β pode ser tratado como uma constante por linguagem, como assumem os multiplicadores fixos, ou varia conforme a natureza do problema (recursão profunda, programação dinâmica, grafos, I/O massivo)?

A Seção 2 apresenta a fundamentação teórica e os trabalhos relacionados; a Seção 3, a metodologia; a Seção 4, os resultados organizados pelas questões de pesquisa; e a Seção 5, as conclusões e os trabalhos futuros.

2. Fundamentação Teórica e Trabalhos Relacionados

Esta seção apresenta os conceitos de avaliação automatizada de código que fundamentam o trabalho e posiciona a contribuição frente à literatura sobre calibração de tempos-limite e equidade entre linguagens.

2.1. Avaliação automatizada de código

Juízes de código online são sistemas projetados para avaliar automaticamente a correção e a eficiência de soluções submetidas [Wasik et al. 2018]. O fluxo é relativamente padronizado: o usuário envia um código-fonte; o sistema compila ou interpreta esse código e o executa em um ambiente controlado com um conjunto de casos de teste predefinidos; durante a execução, verifica-se a saída e se a solução respeita os limites de tempo e memória; por fim, emite-se um veredito. Os principais vereditos incluem Accepted (AC), Wrong Answer (WA), Time Limit Exceeded (TLE), Runtime Error (RTE) e Compilation Error (CE) [Wasik et al. 2018].

Esses sistemas são eficientes para processar submissões em larga escala, garantindo rapidez, padronização e objetividade na avaliação [Ala-Mutka 2005]. Nesse fluxo, a definição do tempo-limite desempenha papel crítico: além de evitar que algoritmos de complexidade excessiva monopolizem recursos, ele atua como mecanismo indireto de verificação da eficiência algorítmica [Laaksonen 2017]. A forma como esse limite é estabelecido tem, portanto, impacto direto sobre o veredito.

As plataformas tradicionais calibram esse limite a partir de soluções de referência otimizadas em C++. Parte aplica o mesmo limite a todas as linguagens, ignorando suas diferenças de execução; outras adotam multiplicadores fixos (tipicamente $2\times$ ou $3\times$ sobre o tempo do C++), que reconhecem a diferença, mas a tratam como um fator constante. Nenhuma das duas abordagens, porém, considera que essa sobrecarga pode variar conforme o algoritmo. Enquanto C++ gera código nativo e explora recursos de baixo nível, Python acumula sobrecarga de interpretação e opera com limites de pilha mais restritos.

2.2. Trabalhos relacionados

Os trabalhos aqui discutidos foram identificados por busca não sistemática nas bases ACM Digital Library, IEEE Xplore e Google Scholar, com termos como *online judge*, *automated assessment* e *time limit*, priorizando surveys e estudos diretamente relacionados à calibração de tempos-limite e à equidade entre linguagens.

Entre as plataformas de programação competitiva, o Codeforces oferece a funcionalidade TimeScaling, na qual o tempo-limite é multiplicado por fatores por linguagem, com teto de 30 segundos¹. Esses fatores, no entanto, são definidos heurísticamente, sem base empírica documentada, e permanecem constantes por linguagem, sem variar com a natureza do problema. Outras plataformas amplamente utilizadas, como LeetCode, AtCoder e CSES, não disponibilizam documentação pública sobre suas políticas de tempo-limite por linguagem, e discussões públicas registram casos em que soluções algorítmicamente equivalentes recebem veredito negativo em Python enquanto são aceitas em C++².

¹Exemplo de grupo que adota essa política: <https://codeforces.com/group/MWSDmqGsZm/contest/219774/extraTimeFactors>

²Exemplo de reporte no canal oficial: <https://github.com/LeetCode-Feedback/LeetCode-Feedback/issues/36357>

Trata-se de uma decisão de alto impacto, já que o veredito de tempo é frequente: Zhang et al. [Zhang et al. 2025] observam que o TLE representa 18% de uma amostra de 2,7 milhões de submissões do Codeforces.

No campo acadêmico, Wasik et al. [Wasik et al. 2018] apresentam uma revisão abrangente de juízes online e suas aplicações educacionais. Revisões sistemáticas recentes sobre avaliação automática em programação [Paiva et al. 2022, Messer et al. 2024] concentram-se na correção via testes e na geração de feedback, sem tratar de como a linguagem escolhida pelo estudante afeta o veredito recebido. Combéfis [Combéfis 2022] classifica técnicas de avaliação de código e observa que poucos estudos comparam empiricamente a avaliação automática e a manual em critérios como consistência e justiça, e que a evidência disponível permanece inconclusiva.

Apesar da diversidade de enfoques, a questão da equidade entre linguagens na definição de tempos-limite permanece pouco explorada nos trabalhos analisados, especialmente a possibilidade de que o fator de calibração apropriado varie conforme a categoria do problema. Este trabalho difere ao quantificar empiricamente essa inequidade e ao propor um mecanismo de calibração automática sensível a essa variação, com foco no contexto educacional.

3. Metodologia

Este trabalho caracteriza-se como um estudo experimental de medição, de natureza explicativa, inserido no paradigma positivista e de abordagem quantitativa, conduzido sem a participação de seres humanos e que, por isso, dispensa apreciação por Comitê de Ética em Pesquisa. Os dados são analisados por estatística descritiva, com medianas, intervalos interquartis e intervalos de confiança estimados por *bootstrap*, sem testes de inferência. A investigação confronta hipóteses sobre a calibração de tempos-limite com medições controladas, organizadas em duas fases. A primeira emprega seis classes de complexidade sintéticas — problemas com ordem assintótica conhecida, de $O(1)$ a $O(2^n)$ — e valida o protocolo de medição em condições em que a complexidade é conhecida por construção. A segunda aplica o protocolo validado a onze problemas reais do repositório CSES, distribuídos em quatro categorias recorrentes no ensino de algoritmos (grafos, programação dinâmica, *backtracking* e recursão), com 56 soluções em C++ e Python submetidas a 186 casos de teste oficiais. Na segunda fase, o veredito sobre a ocorrência de penalização cabe ao juiz oficial, e não às medições locais.

3.1. Soluções de referência e equivalência entre linguagens

Cada problema foi implementado em C++ e Python, em duas versões corretas que diferem apenas no desempenho. A versão otimizada implementa a complexidade assintótica esperada e mede a disparidade entre as linguagens. A ineficiente preserva a mesma saída, mas com complexidade ou constante deliberadamente maior, e serve para verificar a seletividade do mecanismo, ou seja, sua capacidade de continuar reprovando soluções inadequadas. As implementações de cada problema precisam ser equivalentes entre as duas linguagens, de modo que as diferenças de tempo reflitam o modelo de execução, e não a estratégia adotada. Essa equivalência foi assegurada em três dimensões:

- **Estrutural:** preserva a estratégia algorítmica e as estruturas de dados correspondentes (`vector` $\leftrightarrow$ `list`, `priority_queue` $\leftrightarrow$ `heapq`), com política de entrada e saída simétrica entre as linguagens.

- **Semântica:** mantém as mesmas invariantes e a mesma complexidade assintótica, com ajustes documentados quando uma restrição da linguagem o exige, como o limite de profundidade de recursão em Python.
- **Comportamental:** exige que ambas as versões produzam saídas idênticas sobre os casos de teste oficiais de cada problema.

A equivalência foi verificada por análise estática linha a linha, que confirmou a correspondência estrutural e semântica, com registro das invariantes e da complexidade de cada solução, e por execução sobre os casos oficiais, que confirmou a identidade de saída em todo o conjunto. Diferenças impostas por restrições específicas de uma linguagem foram tratadas como variáveis experimentais. As provas de equivalência e as tabelas de mapeamento entre implementações estão disponíveis no repositório.³

3.2. Ambiente e protocolo de medição

O protocolo substitui multiplicadores fixos por uma calibração empírica em ambiente controlado, organizada em torno de quatro elementos: o dimensionamento da escala de entrada, as estratégias de anti-otimização, a repetição adaptativa com critério de convergência e a coleta de métricas em ambiente isolado.

3.2.1. Dominância do custo algorítmico

Para que o *overhead* de inicialização não oculte a diferença entre as linguagens, as entradas foram dimensionadas de modo que o custo algorítmico superasse esse *overhead* em pelo menos uma ordem de grandeza. Essa escolha operacionaliza o princípio de que, em *benchmarks* de desempenho, o sinal de interesse deve dominar o ruído de fundo [Jain 1991, Beyer et al. 2019]. Abaixo desse limiar, o tempo medido reflete o custo fixo de processo, e não a operação algorítmica; é o caso das classes $O(1)$ e $O(\log n)$, que por isso são tratadas como piso de *overhead* e não como classes calibráveis.

3.2.2. Estratégias de anti-otimização

Nas versões deliberadamente ineficientes, o compilador pode remover trechos cujo resultado não é observável, descaracterizando a ineficiência pretendida. Para impedir essa interferência sob $\neg O2$, foram inseridos efeitos colaterais observáveis, como contadores acumulados e escritas controladas, que garantem a execução efetiva da versão ineficiente sem alterar sua lógica.

3.2.3. Repetição adaptativa e convergência

O número de execuções foi definido de forma adaptativa, segundo a estabilidade das medições. Adotou-se o intervalo interquartil (IQR) como medida de dispersão, por sua robustez a *outliers* [McGill et al. 1978], e a mediana como estimador central, preferível à média em *benchmarks* de desempenho dada a assimetria das distribuições de tempo [Jain 1991]. A convergência foi fixada em IQR inferior a 15% da mediana para C++ e a

³<https://github.com/VANDRESSAGALDINOS/adaptive-code-judge-tcc>

20% para Python, diferenciação que reflete a maior variabilidade das linguagens interpretadas. O processo inicia com cinco execuções e reavalia o IQR a cada novo bloco de cinco, encerrando ao convergir ou ao atingir o teto de 35 execuções [Kalibera and Jones 2013]. Em cenários de maior variabilidade, intervalos de confiança de 95% foram estimados por *bootstrap* [Efron 1979].

3.2.4. Ambiente e métricas

As execuções ocorreram em *containers* Docker [Merkel 2014], com imagens de versão fixada (GCC 16.1.0, Python 3.11.15, arquitetura arm64) e cada execução recebeu 1 CPU e 512 MB de memória via *cgroups*. O tempo de execução foi medido por relógio de resolução de microssegundos, restrito à solução, com a compilação e a inicialização do *container* excluídas da medição. O pico de memória e o *overhead* de inicialização foram registrados como métricas de controle, sem integrar o cálculo do fator de calibração. O *overhead* confirma que o custo algorítmico domina a escala de entrada.

3.3. Fator de calibração e tempo-limite adaptativo

O fator de calibração β quantifica a diferença de desempenho entre as duas linguagens para um dado problema. Ele é definido como a razão entre as medianas dos tempos de execução de Python e C++ no maior caso de teste, aquele que mais exige o tempo-limite:

$$\beta = \frac{\tilde{t}_{\text{Python}}}{\tilde{t}_{\text{C++}}} \quad (1)$$

onde $\tilde{t}$ denota a mediana das execuções convergidas (Seção 3.2). Por ser uma razão entre tempos medidos no mesmo ambiente, β isola a penalidade atribuível ao modelo de execução da linguagem.

A partir de β , o tempo-limite adaptativo afrouxa o prazo apenas para a linguagem interpretada, mantendo o C++ como referência. Dado o limite-base T do problema, o limite aplicado a cada linguagem é

$$T_{\text{C++}} = T \quad \text{e} \quad T_{\text{Python}} = \beta \cdot T. \quad (2)$$

O limite-base T é o mesmo prazo de referência aplicado pelo juiz tradicional, fixado em um segundo nos experimentos. Sob esse modelo, o C++ permanece sujeito ao prazo original e apenas o Python recebe a margem proporcional a β : as condições se igualam por aumento de prazo, sem qualquer restrição à linguagem de referência.

Quando um problema admite mais de uma solução de referência correta com desempenho distinto, como as formulações iterativa e recursiva de um mesmo algoritmo de programação dinâmica, mede-se um fator para cada formulação. Como o juiz aplica um único limite por problema, adota-se o maior valor como β operacional, uma vez que um limite inferior penalizaria indevidamente a formulação correta mais lenta. Os fatores restantes são reportados como evidência da variação de β com a estrutura da solução (Seção 3.4).

3.4. Aplicação aos problemas do CSES

A segunda fase aplica o protocolo validado a onze problemas do CSES Problem Set, distribuídos pelas quatro categorias recorrentes em disciplinas de análise de técnicas de algoritmos: grafos, programação dinâmica, *backtracking* e recursão. A seleção exigiu, de cada problema, entrada suficientemente escalável para satisfazer a dominância do custo algorítmico (Seção 3.2), potencial de disparidade entre as linguagens e complexidade representativa do nível trabalhado nessas disciplinas.

A escolha do CSES deve-se à disponibilidade pública dos casos de teste, que viabiliza a reprodutibilidade e permite submeter as soluções ao próprio juiz oficial, e ao fato de ser uma plataforma de referência no contexto educacional e na programação competitiva. A avaliação pelo juiz oficial fundamenta a noção de disparidade: quando uma solução Python de complexidade assintótica adequada recebe *Time Limit Exceeded* (TLE) e a equivalente em C++ é aceita, configura-se o que se denomina TLE injusto, uma reprovação atribuível ao modelo de execução da linguagem, e não à complexidade da solução. O veredito dessa injustiça cabe ao juiz oficial; o *pipeline* local apenas mede o fator β e propõe o tempo-limite adaptativo. A Figura 1 resume o encadeamento: a submissão externa coleta os vereditos no juiz oficial, enquanto a calibração de β , a derivação do tempo-limite e a verificação de seletividade ocorrem localmente.

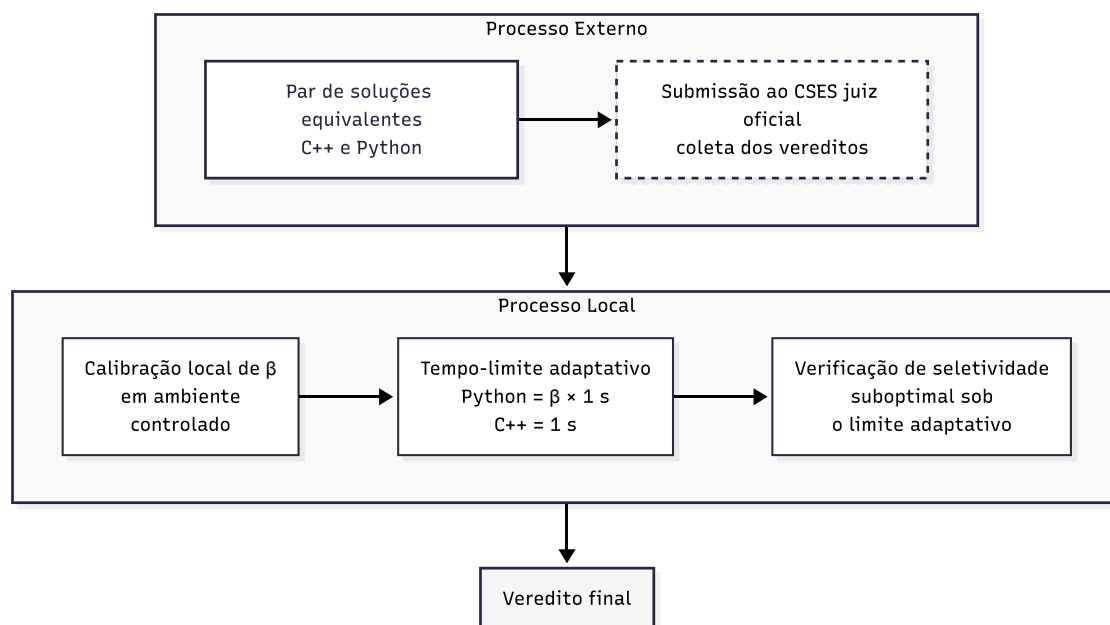


Figura 1. Protocolo de avaliação da segunda fase.

Os problemas cobrem as quatro categorias — grafos, programação dinâmica, *backtracking* e recursão — e abrangem fatores de custo dominante variados, de produtos $n \cdot m$ e laços n^2 e n^3 a percursos em árvore e consultas limitadas por memória, justamente os regimes que mais expõem a diferença entre as linguagens. A cobertura entre categorias não é uniforme, pois o número de problemas que satisfazem todos os critérios varia conforme a categoria, limitação retomada na Seção 3.6.

3.5. Critérios de avaliação

A QP2 é avaliada por dois desfechos: a redução do TLE injusto e a preservação da seletividade contra soluções ineficientes. O TLE injusto é determinado pelo juiz oficial. Para cada problema, contam-se os casos em que a solução Python de complexidade adequada recebeu *Time Limit Exceeded* no CSES enquanto a equivalente em C++ foi aceita, tratando-se o veredito local apenas como evidência complementar, por depender da máquina. A correção é verificada quando o limite adaptativo $\beta \cdot T$ cobre o tempo da solução Python nesses casos, eliminando a reprovação. A análise é descritiva, sem testes de inferência.

A seletividade é avaliada por submissão. Reprova-se a submissão se ao menos um caso excede o tempo-limite. A versão ineficiente é submetida ao limite adaptativo no caso de maior porte e, como o estouro é determinístico por ampla margem, uma execução basta. A seletividade é preservada quando essa versão permanece reprovada sob o limite ampliado, enquanto a de referência é aceita. O problema de controle sem solução a reprovar não participa dessa verificação.

3.6. Ameaças à validade

Construto e validade interna. Atribuir a diferença de desempenho ao modelo de execução depende das soluções comparadas serem equivalentes, o que é sustentado pelo protocolo da Seção 3.1, com verificação estrutural e semântica documentada por problema e identidade de saída sobre todos os casos oficiais. Quanto ao ambiente, não é possível fixar a afinidade de núcleos nem o governador de frequência, e o tempo é medido por relógio *wall-clock*; como β é uma razão entre tempos medidos no mesmo *container*, essas fontes de variação incidem de modo semelhante sobre as duas linguagens e se atenuam na razão, com os efeitos residuais controlados pela repetição adaptativa e pelo descarte de medições não convergentes.

Validade externa e de conclusão. O fator de calibração depende do *hardware* e não é transferível entre máquinas, mas isso é parte do método, que recalibra β por ambiente, e não compromete as conclusões qualitativas. No único problema em que a comparação direta foi viável, o β local convergiu com a razão observada no próprio CSES (35,58 ante 35,5). O conjunto avaliado é moderado — onze problemas reais e seis classes sintéticas —, o que limita a generalização, e a cobertura entre categorias não é uniforme. A inclusão de classes sem disparidade, como o controle e as de piso, mitiga o risco de seleção tendenciosa. A análise descritiva, coerente com o caráter experimental do estudo, deixa a verificação inferencial como trabalho futuro.

4. Resultados e Discussão

Os resultados são organizados segundo as três questões de pesquisa apresentadas na Introdução. Em toda a avaliação, a penalização por tempo é determinada pelo juiz oficial, enquanto ao ambiente local cabe a medição controlada de β e a verificação de seletividade (Seção 3).

4.1. QP1: A inadequação dos multiplicadores fixos

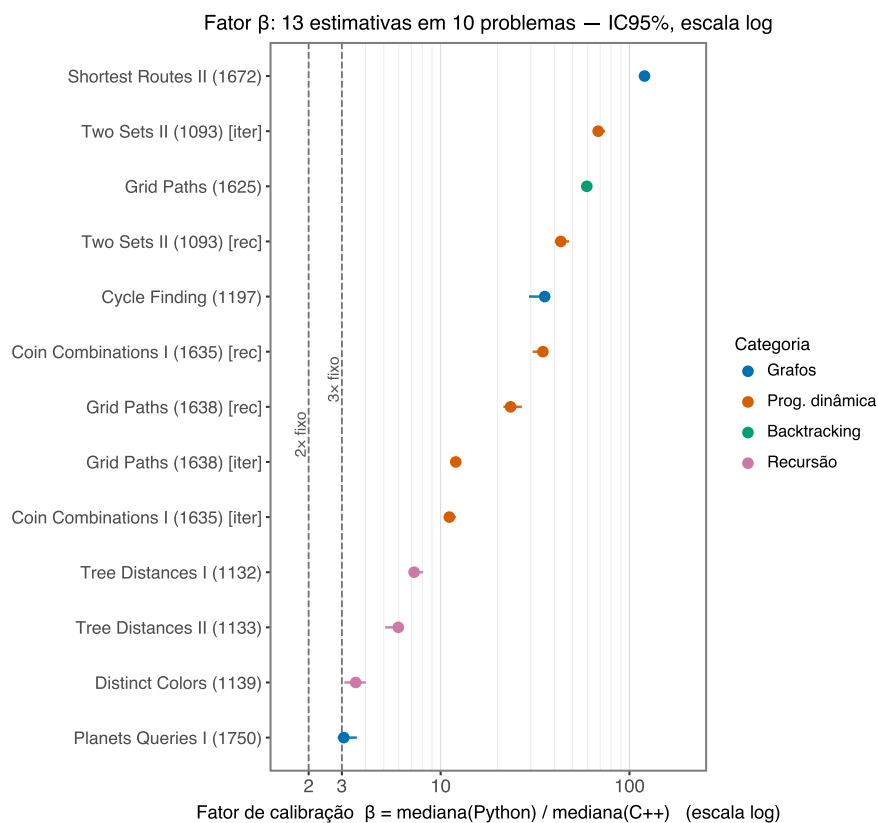
Os multiplicadores fixos de $2\times$ e $3\times$ não capturam a diferença de execução entre as linguagens. A evidência surge já nas classes sintéticas e acentua-se nos problemas reais. Na primeira fase, o fator β não acompanha a ordem de complexidade (Tabela 1).

Tabela 1. Fator β por classe de complexidade sintética (Fase 1), com intervalo de confiança de 95%.

Classe	Natureza da operação	β (IC95%)
$O(1)$	aritmética básica (piso de <i>overhead</i>)	4,3 [4,0–4,5]
$O(\log n)$	busca binária (piso de <i>overhead</i>)	4,1 [3,8–4,4]
$O(n)$	soma de array (varredura linear)	3,9 [3,8–4,1]
$O(n^2)$	soma de matriz (laço duplo)	4,5 [4,2–4,6]
$O(n^3)$	multiplicação de matrizes (laço numérico denso)	77,2 [76,2–79,3]
$O(2^n)$	recursão exaustiva (<i>subset-sum</i>)	33,8 [33,5–34,3]

Entre as classes calibráveis, a soma de array $O(n)$ e a soma de matriz $O(n^2)$ mantêm β próximo de 4. Já a multiplicação de matrizes $O(n^3)$ alcança 77,2, e a recursão exaustiva $O(2^n)$, 33,8 — valores muito acima, embora as ordens de complexidade sejam vizinhas. O que distingue esses casos é a natureza da operação. O laço numérico denso da multiplicação de matrizes é o cenário mais desfavorável ao Python: cada operação passa pelo interpretador, enquanto o C++ compilado a executa em código nativo vetorizado. Na recursão exaustiva, o C++ também paga o custo das chamadas de função, o que reduz a distância entre as duas linguagens. As classes $O(1)$ e $O(\log n)$ ficam abaixo do limiar de dominância de escala e figuram apenas como piso de *overhead*.

Nos problemas reais, o mesmo padrão se confirma de forma mais nítida. A Figura 2 mostra o fator β de cada problema, com intervalo de confiança, em escala logarítmica, junto às linhas de referência dos multiplicadores fixos de $2\times$ e $3\times$.

**Figura 2. Fator β por problema, com IC95% em escala logarítmica.**

Todas as estimativas situam-se à direita dessas linhas, ou seja, o multiplicador fixo subestima a penalidade em todos os problemas, sem jamais superestimá-la. Os valores vão de 3,07 a 120,12: um fator de $3 \times$ seria quase suficiente para Planets Queries I (3,07) e Distinct Colors (3,55), mas subestimaria em cerca de quarenta vezes o exigido por Shortest Routes II (120,12). Por ser constante, o multiplicador fixo não se ajusta à natureza de cada problema, e é justamente essa limitação que a calibração empírica supera.

4.2. QP2: Redução do TLE injusto e preservação da seletividade

Sob o limite tradicional de um segundo, a penalização de soluções Python corretas é frequente, variando de 13% (Grid Paths, 1638) a 56% (Shortest Routes II) dos casos oficiais e atingindo dez dos onze problemas (Figura 3). O controle Chessboard and Queens, de entrada fixa, é o único sem penalização (0 de 10), confirmando que a disparidade depende da natureza do problema e não é um efeito espúrio do experimento.

O limite adaptativo eliminou a penalização sem comprometer a correção. Onde a injustiça também ocorreu no ambiente local, o limite $\beta \cdot T$ cobriu o tempo de todas as soluções Python penalizadas, revertendo a totalidade dos casos sem gerar veredito incorreto. Em ambientes mais rápidos, a solução Python já executa dentro do limite tradicional, de modo que não há injustiça a corrigir, coerente com o método, em que a injustiça é determinada pelo juiz oficial e β se recalibra por ambiente, agindo onde ela ocorre.

A seletividade foi preservada. Nos dez problemas com uma solução ineficiente a ser rejeitada, ela seguiu reprovada por tempo no caso decisivo mesmo sob o limite ampliado, com folga e sem resgate indevido. O detalhamento por problema está no repositório.

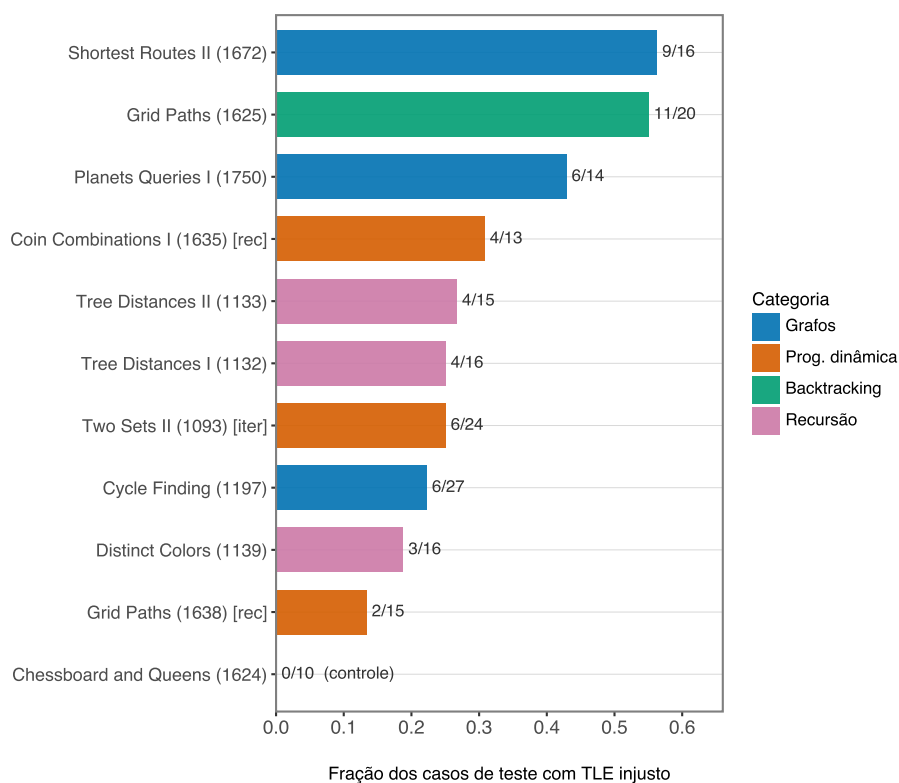


Figura 3. Fração de casos oficiais com TLE injusto por problema, sob o limite tradicional de 1 s no juiz oficial.

4.3. QP3: O fator β varia com a natureza do problema

A terceira questão pergunta se β pode ser tratado como constante por linguagem, como assumem os multiplicadores fixos. Os resultados mostram que não: o fator varia de 3,07 a 120,12 entre os problemas, governado pela natureza da computação, não pela linguagem em si. Dois fatores explicam a dispersão. O primeiro é o tipo de gargalo: problemas limitados por processamento, como o Floyd-Warshall de Shortest Routes II, exibem os maiores fatores (120,12), pois o laço numérico denso penaliza fortemente o interpretador, enquanto problemas limitados por memória, como o *binary lifting* de Planets Queries I, mantêm β baixo (3,07), já que os acessos dispersos penalizam ambas as linguagens. O segundo é o estilo da solução, nos problemas de programação dinâmica com duas formulações, a recursiva e a iterativa produzem fatores distintos (Figura 4), e a recursão chega a multiplicar β por três em Coin Combinations I (de 11,1 para 34,8).

O efeito do estilo não tem direção fixa: em Two Sets II a formulação recursiva apresenta β menor que a iterativa (43,3 contra 68,3), porque o C++ é que fica mais lento ao memoizar uma tabela de grande porte, aproximando as duas linguagens. O fator de calibração, portanto, não é uma propriedade da linguagem, mas da interação entre a operação, a estrutura da solução e o ambiente — precisamente o que um multiplicador constante não tem como representar.

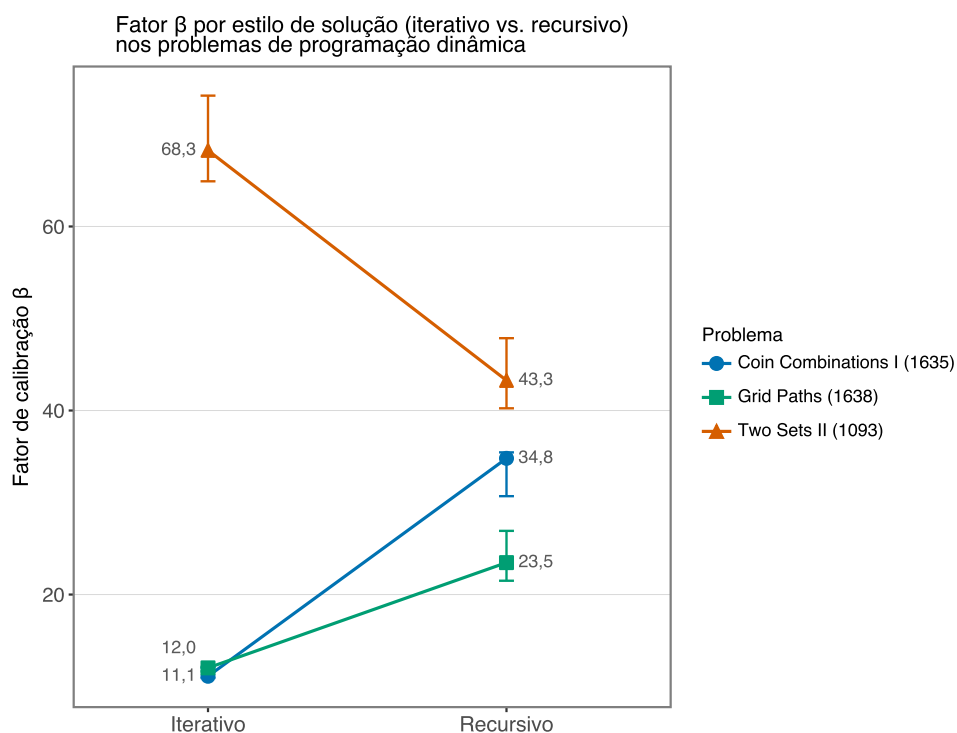


Figura 4. Fator β por estilo de solução (iterativo vs. recursivo) nos problemas de programação dinâmica.

5. Conclusão e Trabalhos Futuros

Este trabalho investigou se a calibração empírica de tempos-limite promove maior equidade entre linguagens em juízes online no contexto educacional. A partir de *benchmarks*

controlados entre C++ e Python, organizados em classes sintéticas e onze problemas reais do CSES, os resultados respondem às três questões de pesquisa. Os multiplicadores fixos de $2\times$ e $3\times$ subestimam sistematicamente a penalidade do Python, ficando aquém do fator exigido em todos os problemas avaliados (QP1). O limite adaptativo $\beta \cdot T$ elimina o TLE injusto sobre soluções corretas sem admitir soluções ineficientes nem respostas incorretas, preservando a seletividade que justifica a existência do tempo-limite (QP2). E o fator de calibração não é uma constante por linguagem, como assumem os multiplicadores fixos, mas varia com a natureza do problema, o tipo de gargalo e o estilo da solução, chegando a inverter de sentido entre formulações de um mesmo algoritmo (QP3).

Em conjunto, as evidências sustentam que calibrar o limite por problema, em vez de aplicar um multiplicador fixo, desloca a avaliação do modelo de execução da linguagem para a correção e a complexidade da solução. Essa mudança alinha o juiz ao propósito pedagógico, ao reduzir penalizações que decorrem da escolha da linguagem e não da qualidade algorítmica da resposta do estudante. A abordagem é, além disso, reprodutível e independente de instrumentação interna do juiz, já que se apoia apenas em medições controladas e na submissão ao próprio juiz oficial.

O escopo desta etapa concentrou-se em onze problemas reais do CSES — com 56 soluções submetidas a 186 casos de teste oficiais — e seis classes sintéticas; o fator de calibração é específico do *hardware*, característica inerente ao método, que se recalibra por ambiente. Como trabalhos futuros, pretende-se estender a cobertura a mais categorias, plataformas e linguagens e integrar o juiz adaptativo à avaliação de uma disciplina real, medindo seu efeito sobre os estudantes.

6. Disponibilidade de Artefatos

Todo o material para reprodução — código-fonte do protocolo de *benchmark*, soluções em C++ e Python, dados de medição e *scripts* de análise — está disponível em repositório público: <https://github.com/VANDRESSAGALDINOS/adaptive-code-judge-tcc>.

Declaração sobre uso de Inteligência Artificial

Os autores utilizaram ferramentas de Inteligência Artificial Generativa (Claude e Claude Code, da Anthropic) como apoio à revisão e ao refinamento da redação do texto e à conferência de artefatos técnicos. As ferramentas atuaram como apoio à verificação conduzida pelos autores, sem substituir o julgamento humano. A concepção da pesquisa, o desenho experimental, a coleta e a análise dos dados e a formulação dos resultados são de responsabilidade integral dos autores.

Referências

- Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102.
- Beyer, D., Löwe, S., and Wendler, P. (2019). Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29.
- Combéfis, S. (2022). Automated code assessment for education: Review, classification and perspectives on techniques and tools. *Software*, 1(1):3–30.

- Efron, B. (1979). Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26.
- Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93, New York, NY, USA. ACM.
- Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, New York.
- Kalibera, T. and Jones, R. E. (2013). Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM)*, pages 63–74. ACM.
- Laaksonen, A. (2017). *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*. Undergraduate Topics in Computer Science. Springer.
- McGill, R., Tukey, J. W., and Larsen, W. A. (1978). Variations of box plots. *The American Statistician*, 32(1):12–16.
- Merkel, D. (2014). Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239).
- Messer, M., Brown, N. C. C., Kölling, M., and Shi, M. (2024). Automated grading and feedback tools for programming education: A systematic review. *ACM Transactions on Computing Education*, 24(1):1–43.
- Paiva, J. C., Leal, J. P., and Figueira, Á. (2022). Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education*, 22(3):1–40.
- Siegfried, R. M., Herbert-Berger, K. G., Leune, K., and Siegfried, J. P. (2021). Trends of commonly used programming languages in CS1 and CS2 learning. In *Proceedings of the 16th International Conference on Computer Science & Education (ICCSE)*, pages 407–412. IEEE.
- Silva, L. L. M. d., Cedraz, V. F., Patrocínio, J. A. d., Delabrida, S., and Fortes, R. S. (2024). Juiz online focado no ensino de programação: uma análise de usabilidade de ferramenta autoral. In *Anais do XXXV Simpósio Brasileiro de Informática na Educação (SBIE)*, pages 1234–1247, Porto Alegre. Sociedade Brasileira de Computação.
- Wang, J., Lin, P., Tang, Z., and Chen, S. (2023). How problem difficulty and order influence programming education outcomes in online judge systems. *Heliyon*, 9(11).
- Wasik, S., Antczak, M., Badura, J., Laskowski, A., and Sternal, T. (2018). A survey on online judge systems and their applications. *ACM Computing Surveys*, 51(1):1–34.
- Zhang, J., Gu, J., Zhang, W., Cambronero, J. P., Kolesar, J., Piskac, R., Li, D., and Shi, H. (2025). A systematic study of time limit exceeded errors in online programming assignments. *arXiv preprint arXiv:2510.14339*.