Rodrigo Ferreira Araújo

Vantagens e Barreiras da IA Generativa como Ferramenta Educacional, de Produtividade e Resolução de Problemas de Lógica de Programação e Engenharia de Software: um Estudo Exploratório da Ferramenta GitHub Copilot

Pesquisa Mista

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Orientador: Pedro Olmo Stancioli Vaz de Melo

Belo Horizonte, Minas Gerais
2024

Sumário

Resumo

O grande impacto do advento da IA generativa trouxe para o mundo de engenharia de software um novo paradigma: o uso da IA como ferramenta de produção de código. Mais especificamente, as inteligências artificiais generativas de texto como o *chatGPT* conseguem produzir trechos de código de qualidades partir de prompts descritivos simples, o que levantou dúvidas entre múltiplas comunidades acerca de qual nível essas novas ferramentas poderiam substituir o próprio programador humano em si, mesmo com suas limitações. Nesse viés, este projeto orientado a computação, com uma atuação empírica e experimentar, visou avaliar quantitativa e qualitativamente a performance de uma promissora ferramenta de IA especializada na geração de código, a extensão de IDEs *GitHub Copilot*, por meio da sua aplicação extensiva na resolução de problemas de diferentes categorias e dificuldades de programação do repositório online *beecrowd*.

O ponto chave é avaliar, de forma exploratória, a sua performance e as estratégias lógico-estruturais empregadas pela ferramenta de IA generativa para desvelar contextos de uso úteis e prejudiciais para o programador humano. Das linguagens usadas para os testes (*JS* e *Python*), os resultados mostraram uma performance acima do esperado para problemas de dificuldade mais fácil e categorias mais simples, mas com uma queda acentuada com o aumento da complexidade geral dos problemas, produzindo mais erros e soluções incompletas. Tanto o *Copilot* quanto o *Copilot Chat*, um *chatbot* dedicado ao uso em codificação, conseguem produzir resultados úteis e bem estruturados em contextos triviais, mas para um contexto mais complexo e multifatorial, introduzem um alto débito técnico e bugs sutis. O trabalho futuro deste projeto consiste em explorar essa IA mais a fundo em outros contextos, mais especificamente em um contexto educacional acerca de engenharia de software e corporativo, como uma revisão da literatura acerca do uso atual da ferramenta em ambiente profissional de codificação.

1 Introdução

Ferramentas de inteligência artificial generativa, tanto de texto como de imagens, músicas e até vídeos, tomaram de assalto a atenção do mundo da tecnologia nos últimos dois anos.

Plataformas como *ChatGPT* para geração de texto em uma abordagem de propósito geral, *Midjourney* para geração de imagens com foco em arte digital e o *GitHub Copilot*[1], que sugere inserções no código sensíveis ao contexto, de rápida evolução em meses e aplicações variadas, trouxeram grande impacto e levantaram múltiplos questionamentos nas suas respectivas áreas de aplicação, seja na produção de textos, imagens ou, como assunto deste trabalho, na geração ou explicação de código.

Nesse sentido, tanto estudantes de programação como programadores de todas as senioridades, perceberam de imediato o ganho de produtividade com uso de ferramentas de IA generativa em suas seções de programação. Principalmente com o uso do *ChatGPT* para sugestões de soluções em código e explicações em mais alto nível bem como o *GitHub Copilot*, uma extensão de IDEs populares como *Visual Studio Code* que, durante o ato de programar, sugere a inserção de trechos de código (em bloco ou em apenas uma linha) com base em informações contextuais do método, classe, arquivo e até mesmo da *codebase* em questão.

O caráter mais específico e "hands-on"do *GitHub Copilot* tornou-o mais visado pelos programadores usuários do VSCode desde o seu lançamento, pois sua proposta fornece uma funcionalidade mais compreensiva e produtiva que as demais ferramentas de IA acopladas no VSCode para geração de código. As duas principais funcionalidades do *Copilot* que o configuram como uma ferramenta poderosa são:

a) Sugestões de código por bloco e/ou *inline* com alta sensibilidade ao contexto do código existente, ou seja, o *Copilot* pode sugerir trechos de código como um método completo com base apenas em uma assinatura previamente escrita ou um comentário no código.

b) *Copilot Chat*: Um chatbot também integrado na IDE com um aspecto conversacional similar ao *ChatGPT*, mas que também fornece comandos especiais a partir de trechos de códigos selecionados, como "explique isso"ou "crie testes para este método".

Apesar da extensão ser acessível apenas via assinatura paga, o GitHub Copilot fornece um plano de acesso gratuito para estudantes universitários.

Portanto, levando em conta as capacidades de geração, compreensão e explanação de código do *Copilot*, infere-se um leque de possibilidades acerca do potencial da ferramenta em contribuir para facilitar não somente o processo de codificação, mas também de aprendizado sob o contexto de conceitos de lógica de programação, estruturas de dados ou então engenharia de software.

O objetivo deste trabalho consiste, por meio de revisão da literatura científica acerca do uso do *Copilot*, análises empíricas e experimentos manuais com a ferramenta, elencar as principais formas de uso que permitem um ganho de produtividade no processo de codificação (e suas limitações associadas), bem como validar e avaliar a performance do *Copilot* na resolução de problemas em diferentes níveis de dificuldade.

[1] https://github.com/features/copilot

2 Referencial Teórico

Nas produções científicas centradas no *GitHub Copilot*, muitas delas, em abordagens empíricas, submeteram a extensão de IDEs à testes de estresse, de modo a validar a robustez dos códigos gerados. Em [1], testes em série com a funcionalidade de sugestão de código são realizados com o *Copilot*, de modo que, a partir somente da especificação textual de 892 métodos em Java, 892 predições de código foram feitas pelo *Copilot*. Em seguida, para cada método foram geradas novas especificações equivalentes às anteriores para serem usadas como base das novas predições de código do *Copilot*, como forma de simular diferentes fraseamentos de diferentes programadores. Desse modo, os resultados indicam certa robustez semântica das sugestões de código do *Copilot* uma vez que, apesar da ferramenta fornecer soluções diferentes em aproximadamente 46% dos métodos, encontraram inconsistências semânticas em 28% dos casos, ou seja, em 28% dos métodos, apenas uma solução correta foi fornecida pelo *Copilot*, seja através da especificação original ou reescrita.

Demais pesquisas que avaliam a produtividade percebida e outros aspectos de experiência do usuário com o uso do *Copilot* apresentam resultados bastante positivos com poucas ressalvas. Em [2], uma *survey* foi realizada com diferentes tipos de usuários do *GitHub Copilot* para levantar estatísticas e *insights* interessantes que cercam o uso da extensão de IDE. Nesse sentido, dentre as descobertas, a pesquisa demonstra um ganho significativo em produtividade percebida entre desenvolvedores de todos os níveis de experiência, desde estudantes até *seniors* com mais de uma década de experiência. A principal métrica avaliada para corroborar esta hipótese foi a taxa de aceitação de sugestões efetivamente mostradas pelo *Copilot*. Além disso, programadores com diferentes níveis de experiência se beneficiam de formas diferentes: desenvolvedores mais iniciantes perceberam um maior crescimento de produtividade e ritmo de codificação, uma melhora na qualidade do código bem como aceitam mais sugestões do *Copilot* e programadores mais experientes filtram mais as sugestões em termos de quantidade e qualidade, mas usam o *Copilot* para trechos de código *boilerplate*, automação de processos rotineiro ou como auxílio para tecnologias desconhecidas. De maneira geral, os programadores reconhecem a corretude inconsistente da extensão, mas é considerado um aspecto secundário à utilidade de uma sugestão como um pontapé inicial para a conclusão de um método/classe/arquivo mais complexo.

Por último, as pesquisas [3] e [4] abordam as capacidades e limitações do *Copilot* em um contexto educacional de uso, de modo que avaliam a performance da ferramenta em resolver problemas e executar tarefas simples de programação, como explicar a solução, escrever testes e consertar *bugs*, bem como avaliam a capacidade da extensão de IDEs de ser uma ferramenta útil para estudantes de programação iniciantes. Os resultados destas pesquisas evidenciam uma performance satisfatória para ações consideradas de "baixo nível", como: completar linhas únicas de código/comentários corretos sintática e semanticamente, escrever testes e resolver atividades introdutórias simples. Contudo, em atividades consideradas de mais alto nível, como escrever um método longo com um design satisfatório, compreensão programática e algorítmica nas sugestões de código e explicações e depuração, de modo que é requerida habilidade do programador em detectar os ocasionais erros sintáticos e lógicos das sugestões aceitas.

3 Metodologia

De modo mais específico, este trabalho explorará as duas principais funcionalidades do *GitHub Copilot* (sugestões de código e *chatbot*) em três frentes: Avaliação da performance da ferramenta na resolução de problemas de programação de múltiplas dificuldades e categorias disponíveis na plataforma de programação competitiva, de modo a verificar a corretude das soluções propostas, quantificar as alterações necessárias para corrigir eventuais erros sintáticos e/ou semânticos e realizar análise quantitativa e qualitativa dos códigos produzidos pela inteligência artificial. O objetivo deste experimento é consolidar um estudo observacional acerca de como operacionalizar melhor esta inteligência artificial generativa especializada para código, que está em versões iniciais de desenvolvimento, atualmente versão 0.17. Experimentação com o *Copilot* e *Copilot Chat* para abordar conceitos, tópicos e exercícios considerados fundamentais em Ciência da Computação, de maneira a considerar a ferramenta como um artefato educacional útil ou não e, por fim, a realização de uma pesquisa focada em artigos/*surveys* centrados em experimentações e análises do *Copilot* para elencar as principais vantagens, casos de uso e limitações da ferramenta para desenvolvedores de diferentes níveis de senioridade.

3.1 Passos Gerais

Os principais passos estão descritos nos seguintes items:

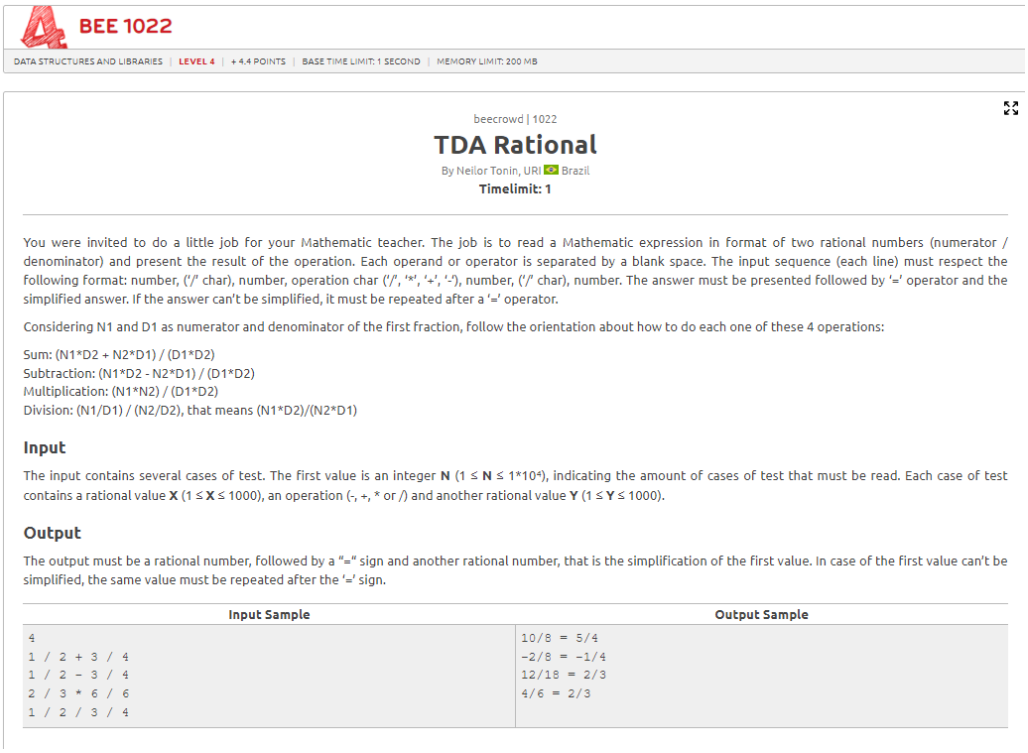
1. Levantamento do conjunto de problemas de uma plataforma agregadora: beecrowd[1].
2. Construção das soluções através do *Copilot*.
3. Análise qualitativa e quantitativa das soluções fornecidas.
4. Levantamento dos tópicos de Ciência da Computação e exercícios associados para atuação do *Copilot*.
5. Aplicação mista (uso prático e teórico) da ferramenta sobre os tópicos e exercícios levantados.
6. Análise dos resultados produzidos.
7. Revisão da literatura acerca do uso do *Copilot* em ambientes profissionais de engenharia de software.
8. Discussão das informações obtidas, como uma revisão sistemática de menor porte, para listar as principais vantagens, casos de uso e limitações da ferramenta nesse ambiente.

[1] https://judge.beecrowd.com

3.2 Avaliação do Copilot na Solução de Problemas Gerais de Programção

A plataforma beecrowd, previamente conhecida como URI Online Judge, similar à plataforma mais conhecida *LeetCode*[2], é uma plataforma agregadora de problemas de programação competitiva que compila problemas de diversas categorias (9 no total), dificuldades e nacionalidades, mas que comumente fornece as descrições dos problemas em inglês ou português brasileiro. Cada problema em si é acompanhado de um *Code Judge*, um campo de texto onde, selecionada uma linguagem de programação, qualquer usuário pode submeter uma solução em código na linguagem selecionada a um programa que a verificará perante múltiplos casos de teste fornecidos para o problema e reporta desde discrepâncias entre as saídas produzidas *versus* as esperadas até erros de compilação/execução.

De forma padronizada, cada problema é identificado por um número, um título, e sua descrição contendo um texto informativo do contexto, exemplo(s) de entrada(s) e respectiva(s) saída(s) esperada(s) e origem do problema, que pode variar desde organizadoras de olimpíadas de programação a usuários anônimos. Ademais, cada problema possui restritivos de tempo e espaço para as soluções submetidas, como, por exemplo, na figura 1,



BEE 1022

DATA STRUCTURES AND LIBRARIES | LEVEL 4 | + 4.4 POINTS | BASE TIME LIMIT: 1 SECOND | MEMORY LIMIT: 200 MB

beecrowd | 1022

TDA Rational

By Neilor Tonin, URI Brazil

Timelimit: 1

You were invited to do a little job for your Mathematic teacher. The job is to read a Mathematic expression in format of two rational numbers (numerator / denominator) and present the result of the operation. Each operand or operator is separated by a blank space. The input sequence (each line) must respect the following format: number, ('/' char), number, operation char ('/', '*', '+', '-'), number, ('/' char), number. The answer must be presented followed by '=' operator and the simplified answer. If the answer can't be simplified, it must be repeated after a '=' operator.

Considering N1 and D1 as numerator and denominator of the first fraction, follow the orientation about how to do each one of these 4 operations:

Sum: (N1*D2 + N2*D1) / (D1*D2)
Subtraction: (N1*D2 - N2*D1) / (D1*D2)
Multiplication: (N1*N2) / (D1*D2)
Division: (N1/D1) / (N2/D2), that means (N1*D2)/(N2*D1)

Input

The input contains several cases of test. The first value is an integer **N** ($1 \le N \le 1*10^4$), indicating the amount of cases of test that must be read. Each case of test contains a rational value **X** ($1 \le X \le 1000$), an operation (-, +, * or /) and another rational value **Y** ($1 \le Y \le 1000$).

Output

The output must be a rational number, followed by a "=" sign and another rational number, that is the simplification of the first value. In case of the first value can't be simplified, the same value must be repeated after the '=' sign.

Input Sample	Output Sample
4 1 / 2 + 3 / 4 1 / 2 - 3 / 4 2 / 3 * 6 / 6 1 / 2 / 3 / 4	10/8 = 5/4 -2/8 = -1/4 12/18 = 2/3 4/6 = 2/3

Figura 1 – Exemplo de um problema na *Beecrowd*

o problema "TDA Rational", da categoria *Data Structures and Libraries* possui um tempo limite de 1 segundo e um espaço de memória limite de 200mb para cada caso de teste. Por último, é atribuído um "level"para cada problema, uma métrica que abstrai a dificuldade de cada um de modo que, começando no nível 5, o nível do problema se aproxima de 1 a medida que a razão do número de soluções corretas fornecidas *versus* o número de tentativas aumenta, e se aproxima de 10 caso contrário. Desse modo, problemas de níveis mais próximos de 1, no escopo deste

[2] https://leetcode.com/

trabalho, serão considerados mais fáceis e problemas cujo nível é mais próximo de 10 serão considerados mais difíceis. Ainda que limitado pela necessidade de um grande número de tentativas para representar bem um nível de dificuldade, este será um parâmetro importante para a avaliação de performance do *Copilot* ao tentar resolver estes problemas. Para mitigar este efeito superficialmente, problemas com mais de 1000 soluções corretas foram preferencialmente selecionados.

Nesse sentido, a partir da base de problemas da *Beecrowd*, estabelece-se um fluxo de trabalho para avaliação quantitativa e qualitativa da performance do *Github Copilot* na solução de problemas de programação de diferentes categorias e dificuldades.

1. Separação de pelo menos 10 problemas de programação, um de cada nível de dificuldade, para cada uma das 8 categorias disponíveis: "Beginner", "Ad-Hoc", "Strings", "Data Structures And Libraries", "Mathematics", "Paradigms", "Graphs", "Computational Geometry".

2. Fornecer a descrição textual do problema, entrada(s) e saída(s) esperada(s) como contexto em arquivos de programação nas linguagens *Python* e *JavaScript*.

3. A partir da descrição textual do problema, em um primeiro momento, produzir uma solução a partir da sugestão de código padrão do *Copilot*, que se assemelha com um *autocomplete* de IDE's, mas que dinamicamente, linha a linha ou função a função, sugere código de acordo com o contexto do arquivo atual.

4. Se a solução produzida no passo anterior, transferida sem alterações para o *Code Judge* do *Beecrowd*, não for aceita, isto é, não passar em todos os casos de teste da plataforma para um problema em específico, novas tentativas serão feitas a partir da funcionalidade do *Copilot Chat*, que, utilizando a descrição textual do problema somado à tentativas anteriores como contexto, produzirá novas soluções a serem avaliadas pelo *Code Judge*.

5. Prompts simples em inglês como "Solve this problem with javascript/python"serão fornecidos para o *chatbot* até que uma solução aceita pelo *Code Judge* seja produzida. Para cada problema, serão feitas no máximo 10 tentativas, contando com a tentativa inicial a partir do *autocomplete*, ou até que as sugestões de código do *Copilot Chat* convirjam, ou seja, situações em que novas tentativas produzem soluções iguais ou equivalentes às anteriores.

6. Prompts alternativos para o *Copilot Chat* podem ser fornecidos a medida em que erros surgem, como, por exemplo, "This solution (is partially correct/exceeds time limit/exceeds memory limit), try another", para, respectivamente, erros de corretude, ultrapassagem do limite de tempo ou de memória.

7. Para cada problema e linguagem de programação da solução (*Python* ou *JavaScript*), serão armazenadas algumas informações estatísticas importantes para a análise quantitativa:

    a) Dificuldade do problema (1, 2, ..., 10).

    b) Categoria do problema, vide passo 1.

    c) Tipo do problema, que servirá como uma "sub-categoria", como, por exemplo, *parsing* de strings na categoria de strings.

d) **T** = Número de tentativas do *Copilot* para produzir uma solução aceita do problema (método padrão (*autocomplete*) + tentativas via *Copilot Chat*).

e) **M** = Número de interferências manuais para consertos pequenos nas soluções produzidas pela IA.

f) **C** = Porcentagem de corretude (0, solução 100% incorreta, até 1, solução 100% correta) do problema caso nenhuma das soluções produzidas seja aceita pelo *Code Judge*, ou seja, porcentagem dos casos de testes aceitos (informação fornecida pela plataforma) pela melhor solução produzida pela IA. **Obs**: para as soluções que convirjam em erros de limite de tempo/espaço, esse erro será reortado na análise e o seu valor de **C** = 0.

g) Uma medida de performance para cada combinação de problema * linguagem definida pela função: $\mathbf{P} = (11 - T - M) * C$, de modo que a melhor pontuação é 10 (Primeira sugestão do *Copilot* é aceita pelo *Code Judge*, sem pequenas correções manuais, ou seja **P** = 10 = (11 - 1 - 0) * 1). A medida de performance para cada problema em específico será a soma das medidas de performance de cada linguagem analisada, neste caso, $P_{Problema} = P_{Python} + P_{JavaScript}$.

Portanto, com estes passos e métricas definidos, uma análise quantitativa exploratória será feita a partir das soluções sugeridas pelo *Github Copilot* para a seleção compreensiva de problemas da plataforma *Beecrowd*.

4 Contribuição

4.1 Resultados

Todo o código de resolução dos problemas, tratamento e visualização das métricas está disponível no repositório https://github.com/RdgFerreira/poc.

4.1.1 Análise Quantitativa

4.1.1.1 Estatísticas Gerais

Por alto, o uso da ferramenta de IA generativa apresentou resultados ligeiramente acima da média (60%) em corretude e erros. Nesse sentido, o experimento mostrou que, a **partir do conjunto total de problemas** do *beecrowd* agregados:

1. 68% foram solucionados corretamente (100% dos casos de teste passaram) via geração de códigos em *Python* e 67% foram solucionados corretamente via *JS* (lembrando que cada problema foi aplicado ao *Copilot* em ambas as linguagens;
2. 49% foram solucionados corretamente sem alterações manuais via *Python*; 50% para *JS*;
3. 22% foram solucionados corretamente na primeira tentativa sem alterações manuais via *Python*; 19% via *JS*.
4. 19% foram solucionados parcialmente (menos 100% dos casos de teste passaram) via *Python* e 17% de soluções parciais via *JS*;
5. 28% foram solucionados corretamente na primeira tentativa (*autocomplete* do *Copilot*) com *Python* e 25% para *JS*;
6. Por fim, 14.6% não foram resolvidos, isto é, não passaram em nenhum caso de teste, usando *JS*, 13.4% não foram resolvidos em *Python*.

4.1.1.2 Performance Média por Categoria

De acordo com a métrica de performance do *Copilot/Copilot Chat*, avaliou-se o desempenho da ferramenta na solução dos problemas de diversas maneiras a seguir:
Na figura 2 temos um desempenho razoável e acima da média (valores no intervalo 6-8) para as cinco primeiras categorias de problemas da plataforma: iniciante, ad-hoc, strings, estruturas de dados, bibliotecas e problemas de matemática. Para categorias de maior nível, tanto em complexidade quanto de tratamento algorítmico, como paradigmas, grafos e geometria computacional, esse desempenho cai vertiginosamente para valores abaixo da média. Para todas as categorias, não há uma diferença significativa de performance entre linguagens de programação na mesma categoria

4.1.1.3 Performance Média por Dificuldade

Já na figura 3, temos uma análise semelhante: performance média reportada para cada dificuldade associada a um problema.
Pode-se observar uma tendência parecida com o gráfico anterior, nesse caso, de queda de performance com o aumento da dificuldade dos problemas, desta vez com uma queda mais suave nos primeiros valores de dificuldade (os mais fáceis), mas ainda com uma métrica acima da média e uma queda mais abrupta ao final. Contudo, nenhum problema de dificuldade máxima (10) foi corretamente ou parcialmente resolvido pelo *Copilot*.

4.1.1.4 Performance Média por Tópico de Solução

Na figura 4, temos um indicativo de performance média por tipos de solução aplicada aos problemas. A plataforma *beecrowd* fornece uma função de "spoilers", que, para cada problema, revela a abordagem esperada e adequada para a solução em código, como, por exemplo, algoritmo de Dijkstra para solução de problemas de caminhos mais curtos em um grafo.
Nesse sentido, destacam-se uma performance acima da média para problemas que envolvem matemática básica e intermediária, formatação, *parsing* e manipulação de strings, e problemas de busca. Performance mediana para baixo em problemas que envolvem paradigmas como programação dinâmica e gulosos. Por fim, uma performance muito baixa para temas mais avançados como caminhamento e busca em grafos (bfs, dfs).

4.1.1.5 Médias de Tentativas do Copilot Por Categoria e Dificuldade

As figuras 5 e 6, respectivamente, mostram os números médios de tentativas de aplicação da ferramenta de IA para resolução dos problemas por categoria e dificuldade:
Ambos demonstram uma tendência crescente suave, ou seja, o número de interferências necessárias para solução (parcial ou correta) ou não dos problemas cresce a medida que a dificuldade e a complexidade das categorias também cresce. Este crescimento é mais uniforme para as dificuldades, mas, na figura de categorias, problemas de categoria matemática quebram levemente a crescente. Normalmente, problemas em *JS* levam mais tentativas nas abordagens de solução, mas não há diferenças significativas entre linguagens nessas análises.

4.1.1.6 Ocorrências de Erro por Categoria

Das soluções que produziram erro, observou-se:

1. 1 erro de ultrapassagem de limite de tempo em *Python* na categoria iniciante;
2. dois erros de na categoria ad-hoc, um de tempo limite excedido em *JS* outro de memória limite excedida em *Python*;
3. dois erros de tempo limite excedido na categoria de strings e estruturas de dados, um para cada linguagem;
4. um erro de limite de memória em *JS* e dois erros de limite de tempo em *Python* na categoria de matemática

5. na categoria de paradigmas, dois erros de tempo limite escedido em *JS* e dois erros de *Python*, um de cada

6. 5 erros de tempo limite excedido na categoria de grafos, 2 em *JS* e 3 em *Python*,

7. por fim, sobre geometria computacional, 4 erros de tempo limite excedido, 3 em *JS* e 1 em *Python*

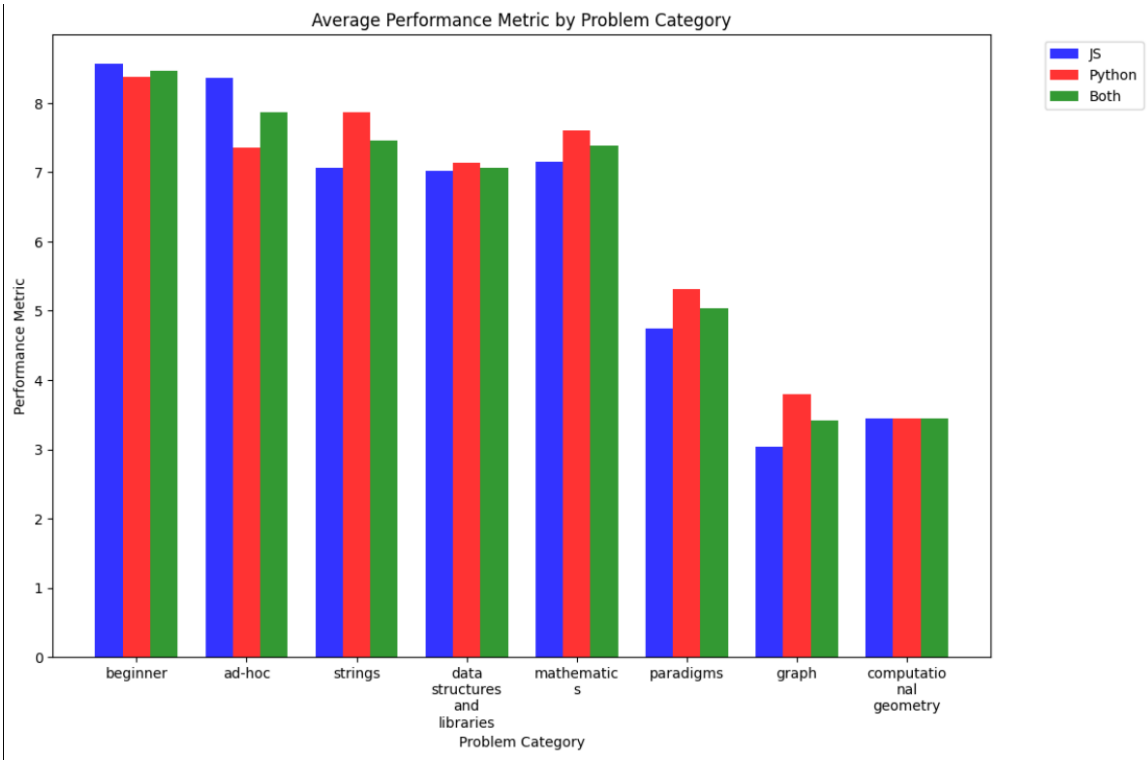


Figura 2 – Performance Média Por Categoria dos Problemas da Beecrowd

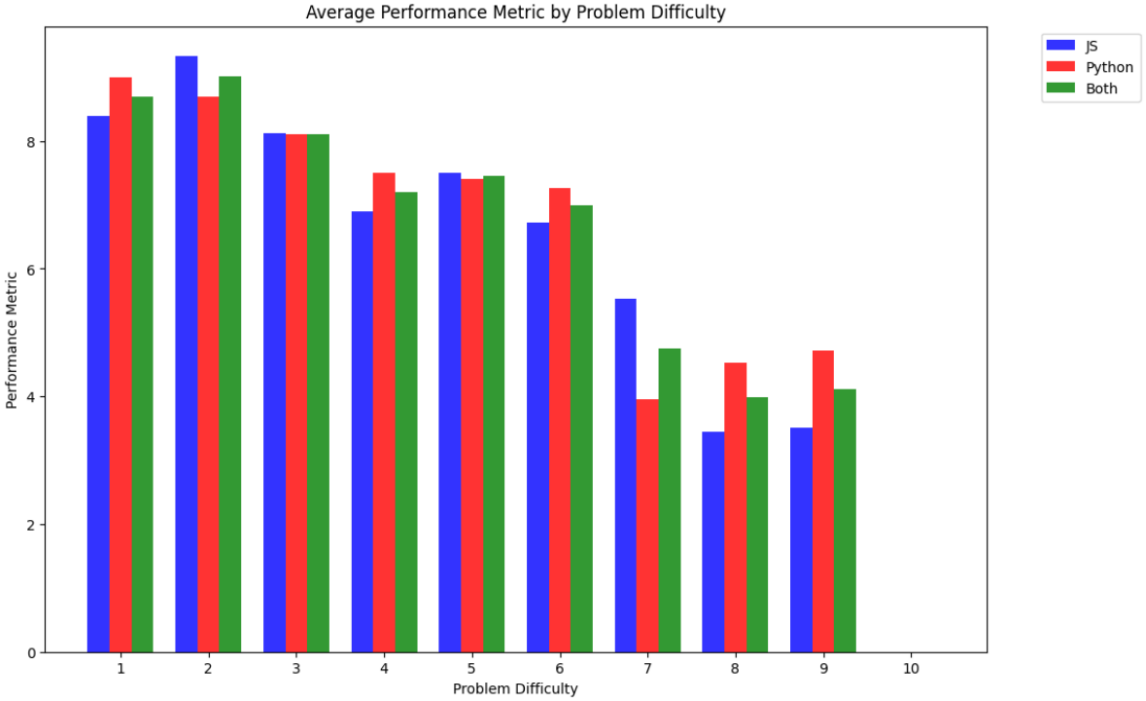


Figura 3 – Performance Média Por Indicativo de Dificuldade da Plataforma

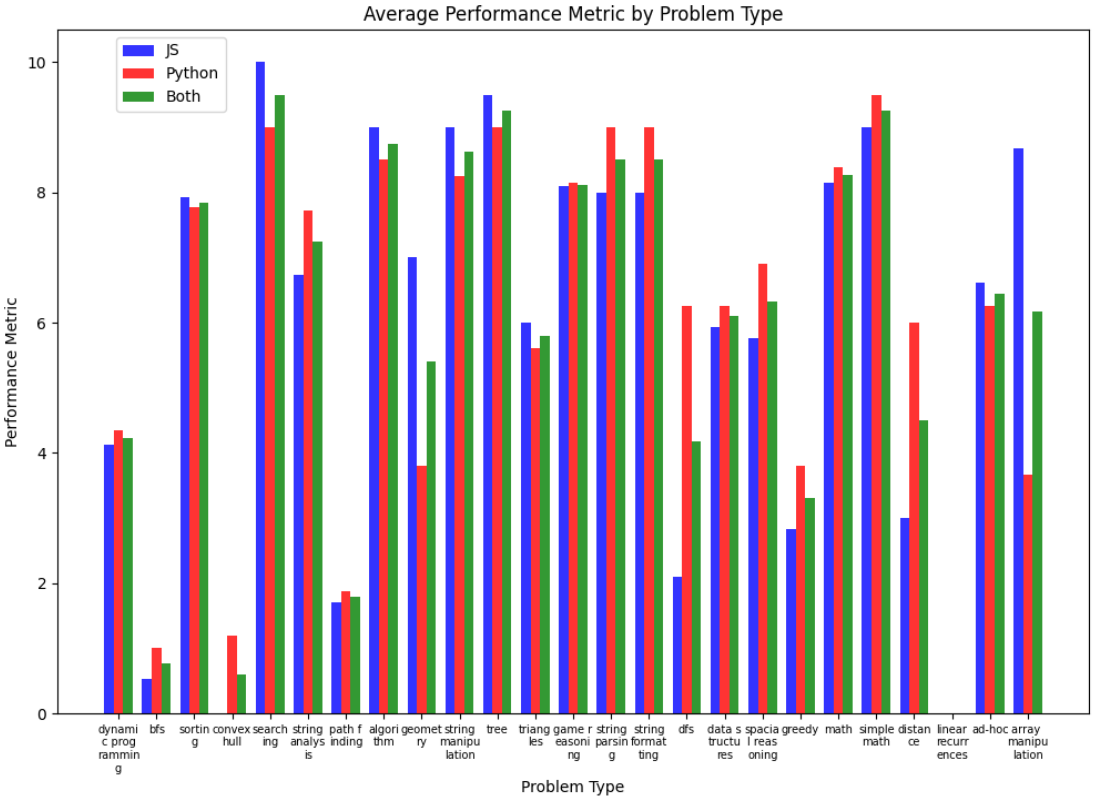


Figura 4 – Média de Performance Por Tipo de Solução

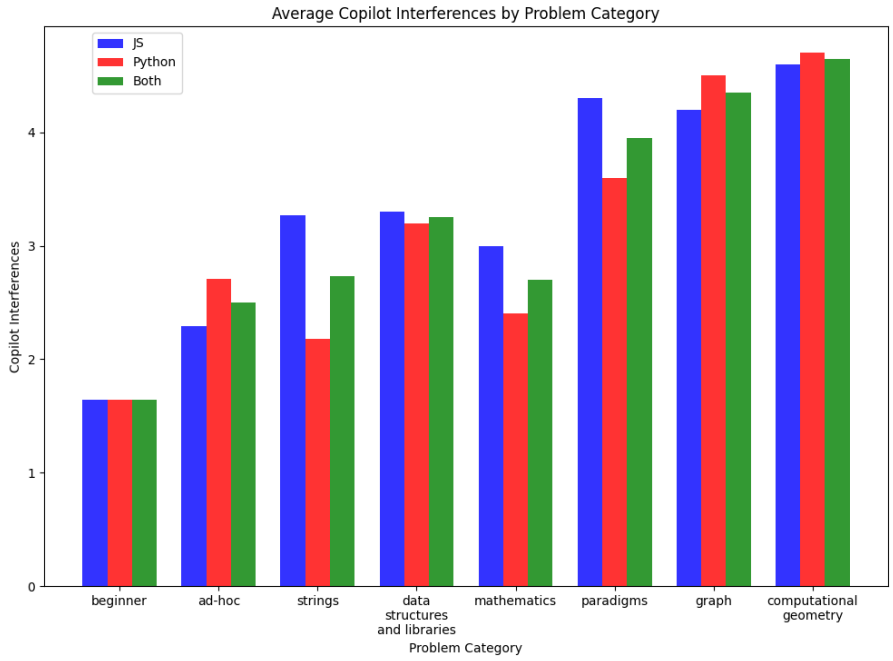


Figura 5 – Tentativas Médias de Resolução dos Problemas por Categoria

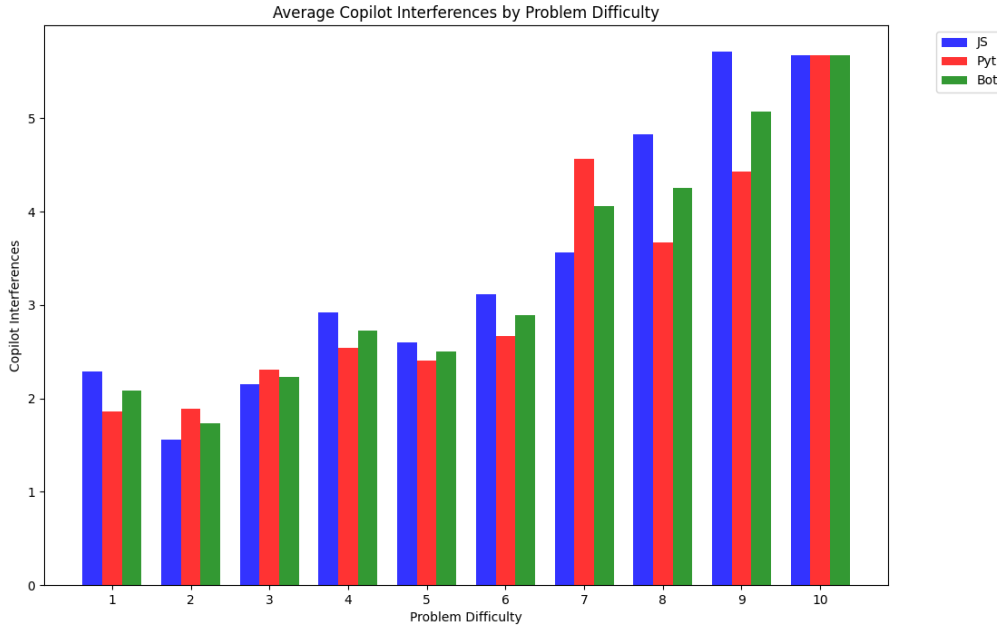


Figura 6 – Tentativas Médias de Resolução dos Problemas por Dificuldade

4.1.2 Análise Qualitativa

À medida que as sugestões de código foram produzidas para os problemas do *beecrowd*, diversas observações lógico-estruturais foram feitas para reconhecer os padrões os quais a inteligência

artificial usa para resolver problemas desta natureza.

4.1.2.1 Pontos Fortes Do Uso do *Copilot*/*Copilot Chat*

Inicialmente, a função padrão de sugestão de código *inline* do *Copilot* encontra sua melhor performance em escopos/contextos reduzidos e/ou não complexos, de modo que sugestões de código de funções auxiliares de tamanho reduzido (algo como ordenar um *array*, encontrar um item em uma lista ou mesmo uma fórmula matemática) ou sugestões de caráter sequencial, como, por exemplo, sugerir "*return result;*"em um contexto de finalização de escrita de um método, estão quase sempre corretas, raramente inconsistentes em lógica, escopo ou sintática. Além disso, o *Copilot* padrão é excelente em capturar contextos lógicos e estruturais do arquivo local e escopo de escrita atual. Nessa lógica, por exemplo, iniciar o escopo de uma função para resolver algum problema de programação com nomes abrangentes como "*solve*"ou simplesmente o título do problema, a sugestão subsequente também será de caráter abrangente, ao passo que terá altas chances da sugestão produzir um código solução para o problema em apenas um método, e.g. Figura 7. Contudo, foi observado que as soluções sugeridas em formato segmentado, ou seja, um código principal contendo uma sequência lógica de funções auxiliares de acordo com o problema a ser resolvido, performam melhor que a abordagem supracitada, ao passo que as sugestões método a método são estruturadas de forma oportuna para a sugestões subsequentes, ou seja, o escopo de referência (lógica e estrutural) para o *Copilot* cresce de forma incremental com a descrição do problema e métodos anteriores, e.g. Figura 8. Desta forma, sendo a IA sensível e acurada ao contexto de escopo e informações textuais do arquivo atual, sugestões comandos estruturais específicos por meio de comentários adicionais nos códigos para a produção de soluções alternativas performam bem, como, por exemplo, "*Solve the problem above reading the input from the standard input and printing the output to the standard output*", bem como o *copilot* tende a repetir padrões estruturais de soluções anteriores (se houver) no mesmo arquivo, de modo que replica sistematicamente os passos padronizados como, por exemplo, dividir o problema em duas funções de lógica do problema e tratamento de input. De maneira geral, somente com o uso da função padrão do *Copilot*, soluções compreensivas, bem estruturadas são produzidas, de modo que, ao observar as sugestões de código, a IA consegue abstrair as propostas gerais dos problemas, desde aplicação de fórmulas matemáticas até operacionalização de diversas estruturas de dados e até mesmo aplicação de algoritmos avançados nas categorias de problemas de grafos e paradigmas.

O *Copilot Chat*, por sua vez, destaca-se por uma vantagem central: versatilidade. Assim, como definido na metodologia, o *chatbot* foi útil para revisões de tentativas subsequentes de soluções após o uso do *Copilot* padrão (*autocomplete*), de modo que, em um mesmo chat, soluções sugeridas podem ser revisadas pelo usuário para melhoria de performance ou simples correções lógicas. Ainda que esteja inclinado à alucinações, especialmente no caso de *chatbots* LLMs [5], a partir da descrição textual como *input* de referência, a IA raramente comete deslizes semânticos ou fuga da proposta nas produções de código de solução.

Por mais, as soluções sugeridas pelo *Copilot Chat* são acompanhadas de pré (Figura 9) e pós-explicações (Figura 10) do código sugerido, de modo que explana o plano de resolução do problema em passos e por fim fornece breves descrições estruturais e funcionais do código

```javascript
function solve(input) {
    const lines = input.trim().split('\n');
    let index = 0;
    const results = [];

    while (true) {
        const [N, M, C, K] = lines[index].split(' ').map(Number);
        if (N === 0) break;

        const serviceRoute = Array.from({ length: C }, (_, i) => i);
        const repairCity = K;

        const graph = Array.from({ length: N }, () => []);
        for (let i = 0; i < M; i++) {
            const [U, V, P] = lines[++index].split(' ').map(Number);
            graph[U].push({ to: V, toll: P });
            graph[V].push({ to: U, toll: P });
        }

        const INF = 1e9;
        const dp = Array.from({ length: N }, () => Array.from({ length: C }, () => INF));
        dp[repairCity][0] = 0;

        for (let i = 0; i < C; i++) {
            for (let u = 0; u < N; u++) {
                for (const { to, toll } of graph[u]) {
                    dp[to][i + 1] = Math.min(dp[to][i + 1], dp[u][i] + toll);
                }
            }
        }

        results.push(Math.min(...dp[serviceRoute[serviceRoute.length - 1]]));
        index++;
    }

    return results.join('\n');
}

// Example usage:
const input = require('fs').readFileSync('/dev/stdin', 'utf8');
console.log(solve(input));
```

Figura 7 – Exemplo de Solução Sugerida em Apenas uma Função

sugerido.

De maneira geral, as soluções produzidas pelo *chatbot* do *GitHub*, ainda que esteja em fases iniciais/experimentais de desenvolvimento, são mais corretas, completas e diversificadas que o *autocomplete* padrão. Nesse sentido, os códigos produzidos possuem mais comentários explicativos e, por vezes, mais bem estruturados. Sob o contexto de resolução de problemas de programação do *beecrowd*, o *Copilot Chat* produz soluções com estruturas de dados e das linguagem mais especializados, como classes e bibliotecas alternativas.

4.1.2.2 Pontos Fracos Do Uso do *Copilot*/*Copilot Chat*

Como toda IA generativa, devido a sua natureza criativa mesmo sob escopos fechados e contextos específicos, o uso do *Copilot* deve ser comedido. Portanto, durante o desenvolvimento das soluções dos problemas, diversos obstáculos técnicos foram encontrados, entre demais performances sub-ótimas.

Primeiramente, tanto o *chatbot* quanto o *autocomplete* falham em capturar o objetivo e detalhes de implementação de problemas com descrições mais longas e/ou complexas, de modo que o

```javascript
function hasAllNumbers(arr) {
    const seen = new Set(arr);
    for (let i = 1; i <= 9; i++) {
        if (!seen.has(i)) return false;
    }
    return true;
}

function isValidSudoku(matrix) {
    // Check rows
    for (let i = 0; i < 9; i++) {
        if (!hasAllNumbers(matrix[i])) return false;
    }

    // Check columns
    for (let j = 0; j < 9; j++) {
        const column = matrix.map(row => row[j]);
        if (!hasAllNumbers(column)) return false;
    }

    // Check 3x3 sub-matrices
    for (let block = 0; block < 9; block++) {
        const subMatrix = [];
        for (let i = Math.floor(block / 3) * 3; i < Math.floor(block / 3) * 3 + 3; i++) {
            for (let j = (block % 3) * 3; j < (block % 3) * 3 + 3; j++) {
                subMatrix.push(matrix[i][j]);
            }
        }
        if (!hasAllNumbers(subMatrix)) return false;
    }

    return true;
}

function sudoku() {
    const input = require('fs').readFileSync('/dev/stdin', 'utf8');
    const lines = input.split('\n').map(line => line.trim()).filter(line => line);
    const n = parseInt(lines.shift(), 10);
    let i = 0;
    for (let instance = 1; instance <= n; instance++) {
        const matrix = [];
        for (let j = 0; j < 9; j++) {
            matrix.push(lines[i++].split(' ').map(Number));
        }
        console.log(`Instancia ${instance}`);
        console.log(isValidSudoku(matrix) ? 'SIM' : 'NAO');
        console.log();
    }
}

sudoku();
```

Figura 8 – Exemplo de Solução Com Funções Sequenciais

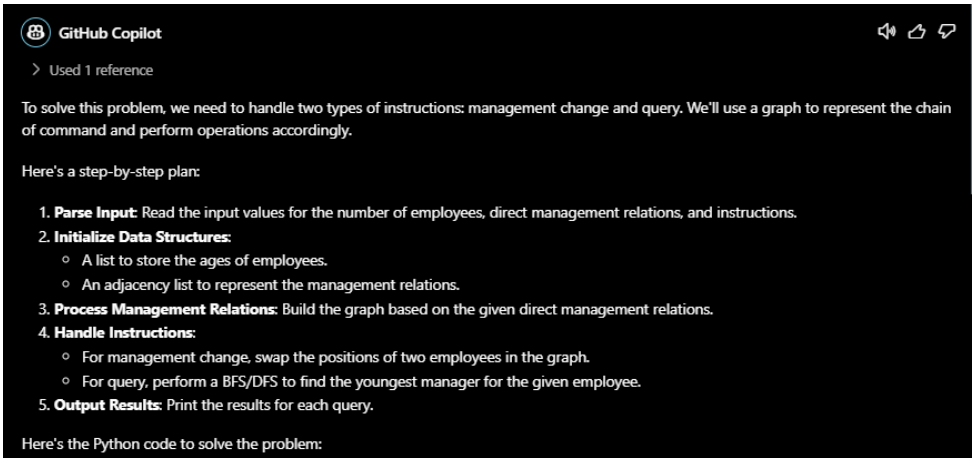


Figura 9 – Explicação Pré Sugestão de Código

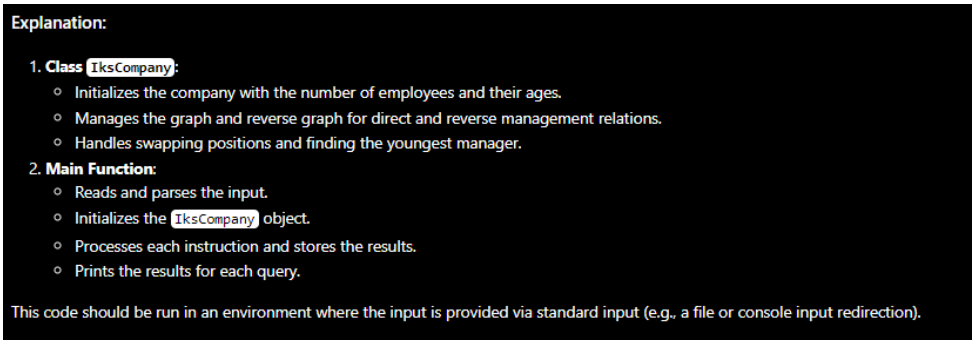


Figura 10 – Explicação Pós Sugestão de Código

efeito é mais percebido na função de *autocomplete* de código. Este efeito acontece especialmente quando um elemento chave da descrição do problema, seja indicação do que produzir como resultado ou como formatar o *input*, esteja descrito em períodos curtos em torno de um grande texto de contextualização/*backstory*.

Ambas as funcionalidades da IA do *GitHub* necessitam de múltiplas iterações de uso para produzir soluções corretas para problemas que envolvam raciocínios espaciais de qualquer natureza. Um dos exemplos claros desta performance baixa pôde ser verificada nas tentativas falhas de resolução do Problema 2785: Pirâmide (*beecrowd*), onde é solicitado que um programa seja desenvolvido para reorganizar caixas em um formato de pirâmide que segue algumas regras específicas. Mesmo após tentativas com sugestões que envolveram o uso de programação dinâmica, a IA apresentou soluções com erros básicos de noção espacial.

Por fim, sobre o *Copilot Chat*, apesar de providenciar soluções robustas e consistentes na maioria dos casos, principalmente para problemas de categorias mais simples e dificuldades mais baixas, tentativas subsequentes de requisição de solução via *chatbot* apresentam pouca variabilidade lógica e estrutural em relação à primeira produzida. Nesse sentido, a IA tende a manter aproximadamente o mesmo formato e lógica aplicados em sugestões de código subsequentes, similar ao efeito verificado na função de *autocomplete* da IA, que, por sua vez, replica estruturas de soluções no contexto do arquivo atual. Esse efeito foi observado mesmo com *prompts* que sugeriram remodular a solução, como posto no passo 6 da seção 3.2.

4.2 Discussão

Com os descobrimentos e elucidações deste estudo, há de se reforçar que o melhor e mais adequado uso da ferramenta no processo de codificação consiste em limitá-la ao propósito que carrega em seu nome: ser uma copilota. A partir de [6] e dos resultados empíricos produzidos neste projeto, se as soluções sugeridas para problemas ou situações de complexidade acima da média não passarem por um filtro analítico de um programador humano, um código, ainda que bem escrito e estruturado, pode ser implantado com cenários *flaky*, *corner cases* desconsiderados e, de forma grave, bugs sutis de alto impacto potencial.

A inteligência artificial generativa aplicada à codificação, em seu estado da arte, já apresenta performance considerável no tratamento de problemas e contextos simples mas desliza cada vez mais a medida que o escopo cresce em complexidade e aborda temas mais avançados. Nesse contexto, o programador sensato, independente de sua senioridade, deve assumir papel de liderança perante o uso da ferramenta, ao passo que programadores iniciantes podem usá-la com mais frequência do que programadores mais experientes mas, universalmente, devem procurar entender plenamente as sugestões de código produzida e usá-las primordialmente como apoio para a implementação final. Portanto, o cenário danoso para o desenvolvimento de software consiste na inversão de papéis no uso do *Copilot*, ou seja, programadores usam as sugestões produzidas como alicerces em larga escala nos projetos, apenas fornecendo acabamentos estruturais e lógicos sem o esforço de documentação e compreensão do código gerado automaticamente.

Em suma, sob o contexto de engenharia/desenvolvimento de software de qualidade, ainda que a tecnologia de IA generativa somente melhorará no futuro, um anteparo humano sempre deve estar presente para balizar as soluções produzidas.

5 Fechamento

5.1 Conclusões

Neste projeto orientado a computação, foi realizado um estudo exploratório de performance das ferramentas de inteligência artificial generativa disponibilizadas pelo *Github Copilot*, em frentes quantitativas e qualitativas, por meio da resolução de problemas de programação fornecidos pela plataforma *beecrowd*. As sugestões de código produzidas performaram relativamente bem em categorias/dificuldades iniciais e intermediárias, ou seja, produziram soluções 100% corretas, mas decaíram com o aumento de complexidade daquelas. Acerca da qualidade das soluções produzidas, tanto a função de *autocomplete* quanto o *chatbot* são plenamente capazes de gerarem funções, classes e demais trechos de código bem estruturados e alinhados com a lógica do problema, mas falham ao produzir bugs sutis e inconsistências mais graves a medida que a complexidade dos contextos e problemas crescem.

5.2 Trabalho Futuro

Experimentação com o *Copilot* e *Copilot Chat* para abordar conceitos, tópicos e exercícios considerados fundamentais em Ciência da Computação, de maneira a considerar a ferramenta como um artefato educacional útil ou não e, por fim, a realização de uma pesquisa focada em artigos/*surveys* centrados em experimentações e análises do *Copilot* para elencar as principais vantagens, casos de uso e limitações da ferramenta para desenvolvedores de diferentes níveis de senioridade.

Referências

[1] MASTROPAOLO, A. et al. On the robustness of code generation techniques: An empirical study on github copilot. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2023. p. 2149–2160.

[2] ZIEGLER, A. et al. Measuring github copilot's impact on productivity. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 67, n. 3, p. 54–63, feb 2024. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/3633453>.

[3] WERMELINGER, M. Using github copilot to solve simple programming problems. In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. New York, NY, USA: Association for Computing Machinery, 2023. (SIGCSE 2023), p. 172–178. ISBN 9781450394314. Disponível em: <https://doi.org/10.1145/3545945.3569830>.

[4] PURYEAR, B.; SPRINT, G. Github copilot in the classroom: learning to code with ai assistance. *J. Comput. Sci. Coll.*, Consortium for Computing Sciences in Colleges, Evansville, IN, USA, v. 38, n. 1, p. 37–47, nov 2022. ISSN 1937-4771.

[5] PATEL, H. Bane and boon of hallucinations in context of generative ai. *Authorea Preprints*, Authorea, 2024.

[6] DAKHEL, A. M. et al. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, Elsevier, v. 203, p. 111734, 2023.

[7] HANSSON, E.; ELLRÉUS, O. *Code Correctness and Quality in the Era of AI Code Generation: Examining ChatGPT and GitHub Copilot*. 2023.

[8] SUNDQVIST, E. *AI-Assisted Unit Testing: Empirical Insights into GitHub Copilot Chat's Effectiveness and Collaborative Benefits*. 2024.