UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JOÃO HEITOR PAES DA SILVA ALBUQUERQUE

ANÁLISE DE ALGORITMOS GENÉTICOS E APRENDIZADO POR REFORÇO PARA GERAÇÃO DE CASOS DE TESTE DE SOFTWARE

RIO DE JANEIRO
2024

JOÃO HEITOR PAES DA SILVA ALBUQUERQUE

ANÁLISE DE ALGORITMOS GENÉTICOS E APRENDIZADO POR REFORÇO PARA GERAÇÃO DE CASOS DE TESTE DE SOFTWARE

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Profa. Anamaria Martins Moreira

RIO DE JANEIRO

2024

CIP - Catalogação na Publicação

A345a

Albuquerque, João Heitor Paes da Silva
Análise de Algoritmos Genéticos e Aprendizado por Reforço para Geração de Casos de Teste de Software / João Heitor Paes da Silva Albuquerque. - Rio de Janeiro, 2024.
54 f.

Orientadora: Anamaria Martins Moreira.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2024.

1. testes automatizados. 2. algoritmo genético. 3. aprendizado por reforço. I. Moreira, Anamaria Martins, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

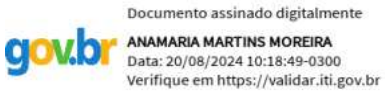
JOÃO HEITOR PAES DA SILVA ALBUQUERQUE

ANÁLISE DE ALGORITMOS GENÉTICOS E APRENDIZADO POR REFORÇO PARA GERAÇÃO DE CASOS DE TESTE DE SOFTWARE

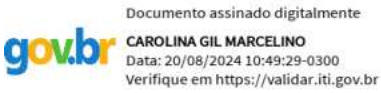
Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 12 de Julho de 2024

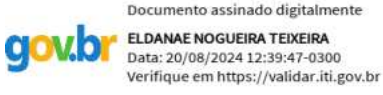
BANCA EXAMINADORA:



Anamaria Martins Moreira
Doutora (IC-UFRJ)



Carolina Gil Marcelino
Doutora (IC-UFRJ)



Eldânae Nogueira Teixeira
Doutora (IC-UFRJ)

AGRADECIMENTOS

Gostaria de agradecer a todos que de alguma forma me auxiliaram na realização deste trabalho, especialmente a professora Anamaria Moreira, que após me introduzir ao mundo de testes de software aceitou me orientar neste trabalho, e minha querida família, Renato, Juliana e Giovana Albuquerque que me apoiaram durante todo o processo.

"It is not the most intellectual of the species that survives; it is not the strongest that survives; but the species that survives is the one that is able best to adapt and adjust to the changing environment in which it finds itself."

Charles Darwin

RESUMO

No artigo "Automation of software test data generation using genetic algorithm and reinforcement learning", os autores Mehdi Esnaashari e Amir Hossein Damia desenvolvem o Memetic Algorithm for Automatic Test case generation (MAAT), um algoritmo genético com aprendizado por reforço que gera casos de teste de software de forma automatizada. Este trabalho tem como objetivo analisar a eficiência da implementação do aprendizado por reforço e das outras modificações aplicadas no MAAT em comparação a um algoritmo genético comum no contexto de códigos de pouca complexidade. Dessa forma, é possível concluir se vale a pena utilizá-lo para automatizar a geração de casos de teste de ferramentas de ensino de introdução à programação. Como o código do MAAT não foi disponibilizado, houve a necessidade de implementar o algoritmo seguindo as instruções relatadas no artigo. Assim, foi possível conduzir experimentos utilizando diferentes códigos de introdução à programação como software a ser testado pelo MAAT e por um algoritmo genético comum para comparar os resultados das suas performances. Analisando os dados desses experimentos, foi possível notar que o MAAT implementado teve eficiência superior em número de iterações necessárias para atingir a cobertura total, tempo de execução do algoritmo e satisfação do critério de cobertura, provando ser um algoritmo adequado para a automatização da geração de casos de teste em ferramentas de ensino de programação.

Palavras-chave: testes automatizados; algoritmo genético; aprendizado por reforço.

ABSTRACT

In the article "Automation of software test data generation using genetic algorithm and reinforcement learning", authors Mehdi Esnaashari and Amir Hossein Damia develop the Memetic Algorithm for Automatic Test case generation (MAAT), a genetic algorithm with reinforcement learning that generates software test cases in an automated way. This work aims to analyze the efficiency of the reinforcement learning and other modifications applied to MAAT in comparison to a common genetic algorithm in the context of low complexity codes. This way, it is possible to conclude whether it is worth to use it to automate the generation of test cases of introductory programming teaching tools. As the MAAT source code was not made available, there was a need to implement the algorithm following the instructions reported in the article. Thus, it was possible to conduct experiments using different introductory programming codes as MAAT's and a common genetic algorithm's software under test to compare their performance results. Analyzing these experiments' data, it was possible to notice that the implemented MAAT had superior efficiency in number of iterations required to achieve full coverage, execution time of the algorithm and satisfaction of the coverage criteria, proving to be an adequate algorithm for the automation of the test case generation in programming teaching tools.

Keywords: automated tests; genetic algorithm; reinforcement learning.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

MAAT	Memetic Algorithm for Automatic Test case generation
CFG	Control Flow Graph
SUT	Software Under Test

SUMÁRIO

1 INTRODUÇÃO

Todo produto, serviço ou sistema desenvolvido em meio a e destinado para um ambiente profissional deve ser projetado visando os propósitos de ser funcional mediante a seu contexto de atuação e satisfatório nos olhos de quem irá utilizá-lo. Sejam por questões de garantia de segurança, otimização de lucro ou outras, existe a necessidade de submeter o que será entregue a um processo de gerenciamento de qualidade, e não é diferente para a criação de um software.

No meio da tecnologia e informática, um conjunto de procedimentos utilizado para verificar se algo atende suas especificações e cumpre sua finalidade é a Verificação e Validação (V&V). Criada em 1998 pelo *Institute of Electrical and Electronics Engineers* e atualizada em 2016, a IEEE 1012 (ELECTRICAL; ENGINEERS, 2017) é um padrão de V&V aplicada a sistemas, softwares e hardwares sendo desenvolvidos, reutilizados ou passando por manutenção. Dentre as etapas incluídas nessa norma, uma delas é o teste do produto.

No caso do teste de software, o mesmo é investigado a fim de fornecer informações sobre sua qualidade em relação ao contexto em que ele deve operar, utilizando o próprio produto para encontrar seus defeitos (AMMANN; OFFUTT, 2016). Dentro do processo de teste de software, existem diversas técnicas possíveis para serem aplicadas dependendo do contexto do software e da situação da equipe de testadores e também diferentes níveis de profundidade do escopo em que o software pode ser testado. Uma dessas possibilidades, o teste unitário é responsável por testar as menores unidades individuais do software, como subrotinas, métodos, classes ou simplesmente pequenos trechos de código. Esse teste irá avaliar a porção de código testada baseado nos dados de entrada fornecidos para o software e suas saídas respectivas esperadas. Essas informações fazem parte dos casos de teste, que são desenvolvidos com o objetivo de alcançarem a cobertura dos requisitos de teste, o conjunto de elementos específicos que os casos de teste devem satisfazer, sendo eles definidos por algum critério de cobertura escolhido para os testes, o conjunto de regras que definem o modo como um código será analisado estruturalmente.

O processo de geração de casos de teste pode ser feito manualmente, porém quanto maior a complexidade do código testado, menos viável essa opção se torna. Idealmente, é desejável criar um conjunto de casos de teste que satisfaça o máximo de requisitos, sendo este então um problema de otimização. Uma das técnicas computacionais utilizadas na tentativa de solucionar esse tipo de problema é a aplicação de algoritmos evolutivos. Na área de testes especificamente, utilizam-se algoritmos genéticos para a geração de casos de teste de software de forma automatizada. No meio acadêmico, diversos autores (SRIVASTAVA, 2009), (BERNDT et al., 2003), (SHARMA; PATANI; AGGARWAL, 2016), (ALANDER; MANTERE; TURUNEN, 1997), (BERNDT; WATKINS,

2005), (BOUCHACHIA, 2007), (DAMIA; PARVIZIMOSAED, 2021), (ESNAASHARI; DAMIA, 2021) publicam artigos relatando suas implementações próprias de algoritmos genéticos com novas modificações comprovando atráves de resultados de experimentos como seus algoritmos genéticos são mais eficientes nas situações testadas.

Já no meio da educação, professores de cursos introdutórios à programação podem utilizar ferramentas para auxiliar na correção de tarefas. Uma delas, a Machine Teaching[1], desenvolvida na UFRJ, possibilita que educadores publiquem enunciados descrevendo desafios lógicos a serem resolvidos por códigos desenvolvidos pelos alunos e esses códigos serão testados fornecendo um conjunto de dados e comparando as saídas com o esperado. Este trabalho pertence a um conjunto de trabalhos de um projeto maior voltado à exploração de opções de melhorias para a ferramenta do Machine Teaching. Nessa ferramenta, a pessoa que propõe cada problema é quem define a função que gera o conjunto dos dados de entrada para testar os códigos soluções desenvolvidos pelos alunos. Sendo assim, a qualidade dos testes será diferente para cada problema e não há garantia de que o conjunto de casos de teste irá atingir um percentual de cobertura satsifatório do código. A utilização de um algoritmo genético como função geradora dessa ferramenta, além de suprir a necessidade da definição de uma função diferente para cada problema, pode otimizar a geração dos seus casos de teste, aperfeiçoando o aprendizado desses alunos. Mais do que isso, existe a possibilidade de que algoritmos genéticos modificados possam ter uma performance ainda melhor.

Na intenção de confirmar tal hipótese, foi realizada uma pesquisa sobre implementações de algoritmos genéticos as quais tentam maximizar a cobertura de requisitos na geração de casos de teste. Dentre algumas estudadas, foi selecionado o algoritmo MAAT (ESNAASHARI; DAMIA, 2021) pela ideia promissora de unir uma técnica de aprendizado por reforço, o Q-learning, com as etapas de busca adaptativa dos algoritmos genéticos. Além disso, o MAAT foi desenvolvido para gerar casos de teste segundo o critério de cobertura por grafos. No trabalho "Modelagem de testes de software: uma análise dos resultados de testes em exercícios de programação"(ALBUQUERQUE; BOECHAT; COUTINHO, 2023), onde foi analisada a aplicação de diferentes técnicas sistemáticas de projeto de software para elaboração de testes para o Machine Teaching, o critério de cobertura por grafos obteve os melhores resultados. Dessa forma, o algoritmo foi implementado e, através de experimentos utilizando códigos de soluções de problemas de introdução à programação como softwares a serem testados, ele foi comparado com um algoritmo genético comum, o qual possuía apenas as etapas básicas de avaliação, mutação, recombinação e seleção. Os dois algoritmos foram então comparados nos quesitos de tempo de execução, número de iterações realizadas e percentual de cobertura obtido. O objetivo deste trabalho é analisar se as modificações implementadas no MAAT são essenciais para uma eficiência significativa na geração de casos de teste para códigos de

[1] https://www.machineteaching.tech

introdução à programação ou se um algoritmo genético comum consegue performar em um mesmo nível.

Ao longo deste trabalho serão discutidos os fundamentos da teoria utilizada como base no Capítulo 2, os objetivos e planejamento dos experimentos, a implementação do algoritmo, um exemplo de utilização dele e possibilidade de integração com a Machine Teaching no Capítulo 3, a execução dos experimentos no Capítulo 4 e a conclusão resultante dos experimentos no Capítulo 5.

2 FUNDAMENTAÇÃO TEÓRICA

Ao longo desse trabalho serão mencionados alguns termos técnicos cujo conhecimento será necessário para a compreensão dos próximos capítulos.

2.1 PROJETO DE TESTES

Com o rápido crescimento da dependência de sistemas de software na maioria dos seguimentos econômicos nas últimas décadas, acompanhado da evolução da complexidade desses sistemas, tornou-se cada vez mais necessário uma forma de garantir um nível de confiabilidade ao software desenvolvido. Com o intuito de suprir essa necessidade, foram articuladas técnicas de verificação e validação, uma delas sendo o uso de testes de software. Para que esses testes fossem eficientes e eficazes, surgiram as metodologias de projeto de testes de software.

Nesta seção serão explorados alguns termos pertinentes ao contexto de projeto de testes de software, inicialmente de forma mais geral e posteriormente mais específico ao tipo de teste realizado neste trabalho.

2.1.1 Conceitos básicos

Terminologia utilizada:

a) Teste de unidade: teste de software com objetivo de testar as menores unidades individuais de código, como métodos ou funções;

b) Critério de cobertura: conjunto de regras que definem o modo como um código será analisado, ou seja, se a estrutura do modelo projetado para representar o software será baseada no domínio dos possíveis valores de entrada, em grafos, expressões lógicas ou sintaxe, para assim definir os requisitos de teste a serem satisfeitos para que os testes sejam concluídos;

c) Requisitos de teste: conjunto de propriedades que o conjunto de casos de teste deve satisfazer ou cobrir;

d) Caso de teste: descrição detalhada de um conjunto específico de valores de parâmetros de entrada, condições de execução e resultados esperados para validar uma funcionalidade específica do sistema; quando usado no contexto de geração automatizada, o termo refere-se especificamente apenas aos valores dos parâmetros de entrada.

   Dado o conjunto de requisitos, formulados baseado no critério de cobertura escolhido para o teste, são projetados os casos de teste para satisfazer esses requisitos.

Se a entrada descrita no caso de teste, respeitando as condições de execução, produz a saída esperada, o código passou o teste para esse caso. Se o código passa o teste para todo o conjunto de casos de teste, satisfazendo todos os requisitos e atingindo a cobertura total do critério, considera-se que não foram encontrados defeitos a serem corrigidos no código.

2.1.2 Critério de cobertura em grafos

Uma das possibilidades de estruturas em que o modelo do software pode ser baseado para a definição dos requisitos de teste. Nesse tipo de critério de cobertura, o código será modelado com base em um grafo, uma estrutura que consiste em um grupo de vértices, ou nós, representando objetos individuais de um conjunto, e arestas, sendo elas um subconjunto de pares de vértices, representando uma relação entre os objetos.

No caso do critério de cobertura em grafos, é utilizado um tipo de grafo específico para a representação do software, o grafo de fluxo de controle. Nele, cada nó representa um trecho sequencial do código que sempre será executado em conjunto e cada aresta uma possibilidade de seguimento do fluxo da execução ao se deparar com uma estrutura condicional. A partir dele, é possível identificar todos os caminhos possíveis de execução do código. O processo de modelagem do grafo a partir do código pode ser feito de forma manual, como foi no caso deste trabalho, porém para códigos de maior complexidade é recomendado a utilização de ferramentas de criação de grafo de fluxo de controle, como o Visustin[1] ou, especificamente para Python, a linguagem utilizada na implementação do algoritmo deste trabalho, o py2cfg [2].

A função apresentada na Figura 1 foi utilizada como exemplo no artigo do algoritmo analisado neste trabalho. Ela recebe como parâmetros três números inteiros e classifica um triângulo que tivesse lados com essas medidas, sem checar sua condição de existência.

```
def triangle(x: int, y: int, z: int):
    #assume-se que os valores de entrada correspondem
    #aos lados de um triângulo válido
    if x != y and x != z and y != z:
        print("scalene")
    else:
        if x == y and y == z:
            print("equilateral")
        else:
            print("isosceles")
```

Figura 1 – Código da função triangle

[1] https://www.aivosto.com/visustin.html
[2] https://pypi.org/project/py2cfg/

Para essa função, por exemplo, o grafo de fluxo de controle, ou CFG (*control flow graph*), poderia ser modelado como apresentado na Figura 2. Nela, o primeiro nó representa o início da função com a primeira estrutura condicional, levando a uma bifurcação com arestas conectando aos nós seguintes. O segundo nó representa a condição sendo satisfeita (caso de um triângulo escaleno). Já no terceiro nó, onde a condição é falsa, há uma segunda estrutura condicional com mais uma bifurcação de arestas. Dessa vez, o quarto nó é a condição verdadeira (caso de um triângulo equilátero) e o quinto nó o caso restante (triângulo isósceles). O segundo, quarto e quinto nó possuem arestas que permitem o caminho chegar no sexto nó, o final do grafo, visto que após chegar em qualquer uma dessas três situações a função termina sua execução.

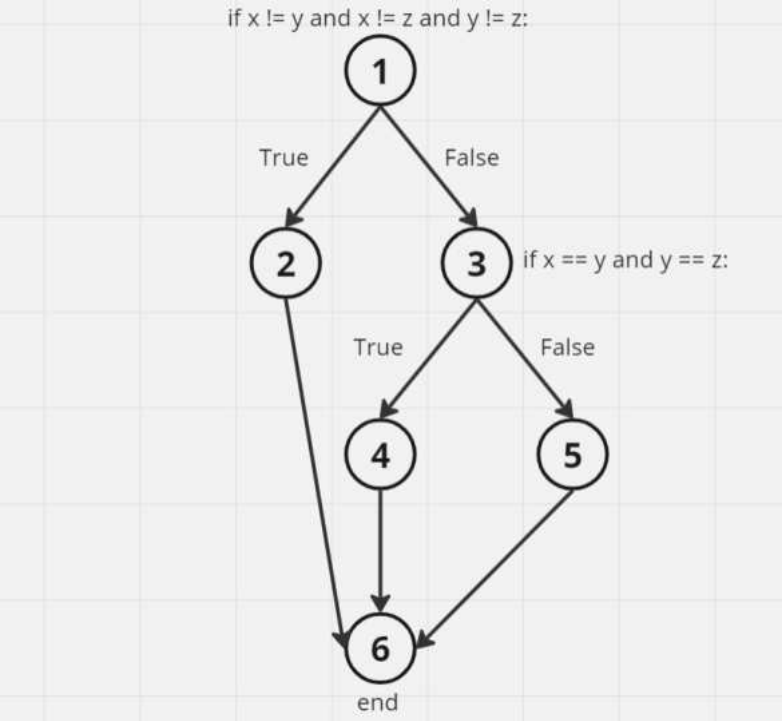


Figura 2 – CFG da função triangle

Para a definição dos requisitos de teste, é necessária a escolha do tipo específico de cobertura em grafos.

- Na cobertura por nós, os requisitos são satisfeitos se após a execução de todos os casos de teste todos os nós do grafo de fluxo de controle tiverem sido visitados, ou seja, se todos os conjuntos de trechos de código que cada nó representa tiverem sido executados pelo menos uma vez durante o teste. Para a CFG da função triangle por exemplo, o conjunto de requisitos de teste seria: {1,2,3,4,5,6}.
- Já na cobertura por arestas, a ideia é parecida, mas em vez de todos os nós visitados é necessário que todas as arestas tenham sido caminhadas. Para a CFG da função

triangle por exemplo, o conjunto de requisitos de teste seria: {(1,2),(1,3),(2,6),(3,4),(3,5),(4,6),(5,6)}.

- Na cobertura por par de arestas, em vez de arestas individuais, é necessário que o conjunto de requisitos de teste contenha todos os possíveis pares de arestas sequenciais, ou seja, de forma que exista um caminho entre o primeiro e terceiro nó passando pelo segundo nó. Para a CFG da função triangle por exemplo, o conjunto de requisitos de teste seria: {[(1,2),(2,6)],[(1,3),(3,4)],[(1,3),(3,5)],[(3,4),(4,6)],[(3,5),(5,6)]}.
- Finalmente, na cobertura por caminhos, cada requisito de teste será um caminho possível de ser executado no grafo de fluxo de controle. Para a CFG da função triangle por exemplo, o conjunto de requisitos de teste seria: {(1,2,6),(1,3,4,6),(1,3,5,6)}.

Como funções com laços podem possuir CFGs com infinitos caminhos dependendo de quantas vezes cada laço seja executado, o conjunto de caminhos dos requisitos seria ilimitado. Por exemplo, na CFG de uma função que possui um laço que se inicia no primeiro nó e termina no segundo nó, assim como mostrado na Figura 3, se não houvesse um limite de vezes que esse laço poderia ser repetido, o conjunto de requisitos do critério de cobertura por caminhos seria: {(1,2,3),(1,2,1,2,3),...,(1,2,1,...,2,3)}.

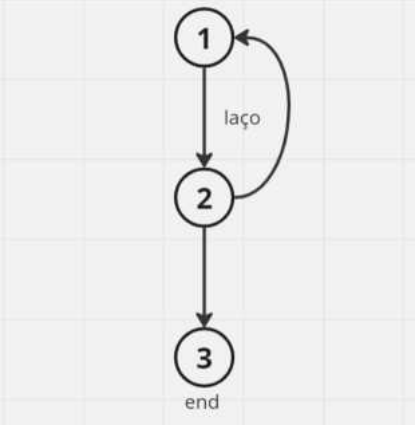


Figura 3 – CFG de uma função com laço

Existem algumas soluções possíveis para a resolução desse problema, uma das possibilidades sendo ao invés de considerar os caminhos como requisito, usar o conjunto de nós de cada caminho, pois mesmo que o conjunto de caminhos seja infinito o conjunto de nós de cada caminho é limitado. Dessa forma, visando facilitar a análise da cobertura do grafo, para este trabalho foi utilizado o critério de cobertura por conjuntos de nós em caminhos, de forma que o conjunto de requisitos de teste contém todos os caminhos possíveis, porém sempre ordenados e sem repetições de nós. No caso de possíveis laços, dois caminhos de execução do grafo que visitarem exatamente o mesmo conjunto de nós, mesmo que em ordens e/ou quantidade de vezes diferentes, serão considerados para a

satisfação do mesmo requisito. Para a CFG da função genérica com laço por exemplo, o conjunto de requisitos de teste seria: {(1,2,3)}.

2.2 ALGORITMOS GENÉTICOS

No âmbito da ciência da computação, especificamente na tentativa de solucionar problemas de busca e otimização, foram desenvolvidos os algoritmos evolutivos, um conjunto de meta-heurísticas inspiradas no processo de seleção natural. Dentre eles existem os algoritmos genéticos, que utilizam operadores inspirados no princípio Darwiniano de reprodução e sobrevivência dos mais aptos: a mutação, a recombinação e a seleção (GOLDBERG, 1989). Esse algoritmo pode ser utilizado para a geração automatizada de casos de teste de software.

Um algoritmo genético comum costuma funcionar de forma que a população é gerada de forma aleatória e, logo após, avaliada seguindo algum critério com o propósito de ranquear seus cromossomos baseado na aptidão deles em relação a tal critério. Depois disso, são escolhidos aleatoriamente cromossomos pais para serem recombinados e gerarem novos cromossomos e alguns cromossomos, dependendo da probabilidade de mutação, terão parte de seus dados alterados. Finalmente, seguindo alguma regra preestabelecida baseada nos valores de aptidão, são selecionados cromossomos de uma parte da população atual para seguirem para a geração seguinte, enquanto o resto é descartado e substituído por novos cromossomos gerados de forma aleatória para, junto dos que sobreviveram, compor a nova geração da população, que passará por todo o processo mais uma vez, se repetindo até que as condições de parada definidas para o algoritmo sejam atendidas.

No caso de algoritmos genéticos sendo utilizados para a geração de casos de teste de um software, existem diversas possibilidades para o formato da estrutura dos cromossomos, uma delas dependendo de quantos parâmetros a função testada recebe e quantos requisitos de teste serão definidos pelo critério de cobertura escolhido. Algoritmos genéticos podem ser projetados para usar diferentes tipos de critérios de cobertura. No caso do algoritmo estudado neste trabalho, a cobertura utilizada é baseada em grafos para poder definir os requisitos de teste a partir da CFG do código. A forma como os cromossomos têm sua aptidão avaliada também depende da escolha de critério de cobertura. Uma possibilidade possui a aptidão representando o percentual dos requisitos satisfeitos. No caso da utilização de um algoritmo genético para gerar casos de teste para a função triangle mostrada anteriormente, considerando algum critério de cobertura por grafos, cada cromossomo da população seria um possível conjunto de casos de teste, tendo um caso para cada requisito de teste, e cada caso de teste contendo três números inteiros, como ilustrado na Figura 4.

Figura 4 – População de possíveis cromossomos para a função triangle

2.2.1 Conceitos básicos

Terminologia utilizada:

a) População: conjunto de cromossomos;

b) Cromossomo: conjunto de parâmetros que define uma proposta de solução para o problema a ser resolvido pelo algoritmo genético;

c) Aptidão: valor númerico relativo à qualidade do cromossomo definido após uma avaliação;

d) Mutação: processo de alteração aleatória de partes do cromossomo;

e) Recombinação: processo onde um par de cromossomos é selecionado e cada um dividido em duas partes (não necessariamente de mesmo tamanho) para gerar dois novos cromossomos compostos pela junção da primeira parte de um dos cromossomos do par selecionado e a segunda parte do outro;

f) Seleção: processo de escolha de quais cromossomos da população atual farão parte da próxima geração baseada na aptidão deles.

2.3 ALGORITMO MAAT

O Memetic Algorithm for Automatic Test case generation (MAAT) é um algoritmo genético gerador de casos de teste desenvolvido por Mehdi Esnaashari e Amir Hossein Damia e descrito no artigo "Automation of software test data generation using genetic algorithm and reinforcement learning" de 2021 (ESNAASHARI; DAMIA, 2021). Uma das grandes modificações que diferem o MAAT de algoritmos genéticos padrão é a implementação do Q-learning, um algoritmo de aprendizagem por reforço. Ademais, o algoritmo também implementa uma busca adicional e a ideia de apenas realizar alterações em casos de teste redundantes. O algoritmo, junto a essas modificações citadas, serão explorados de forma mais detalhada na seção 3.2.

3 METODOLOGIA

Neste capítulo, serão re-explorados os objetivos do trabalho, agora com maior entendimento da teoria por trás, esclarecendo melhor o contexto do planejamento realizado em antecipação aos experimentos executados e também explicada detalhadamente a implementação do algoritmo genético utilizado. Além disso, será demonstrado o passo a passo de um exemplo da utilização do MAAT, desde o enunciado do problema publicado na Machine Teaching até o conjunto de casos de teste gerados pelo MAAT para o código solução do problema. Finalmente, será apresentada uma possível maneira de integração do algoritmo à ferramenta de ensino.

3.1 OBJETIVOS E PLANEJAMENTO DOS EXPERIMENTOS

Como citado anteriormente, este trabalho possui como objetivo verificar a eficiência de modificações implementadas a partir de um algoritmo genético comum no âmbito da geração de casos de teste para códigos de introdução à programação. A finalidade dessa análise é de evidenciar se a utilização de alguma dessas variações no lugar de um algoritmo genético comum produz resultados substanciais nesse contexto de geração de casos de teste com códigos menos complexos ou se nessas situações não existe a necessidade de ir além do básico.

Para isso, foi realizada uma pesquisa na busca por artigos acadêmicos que propusessem soluções baseadas em algoritmos genéticos com o objetivo de otimizar a geração automatizada de casos de teste. Dentre os artigos encontrados, foram selecionados inicialmente oito que pareceram promissores. O primeiro (SRIVASTAVA, 2009) implementava um método de identificação dos caminhos de cluster mais críticos na CFG de um software para poder testar primeiro os caminhos mais propensos a manifestarem erros. O segundo (BERNDT et al., 2003) apresentava a ideia de calcular o valor de aptidão de cada cromossomo baseado na sua semelhança com cromossomos das gerações anteriores que apresentaram erros no código e a severidade desses erros. O terceiro (SHARMA; PATANI; AGGARWAL, 2016) detalhava uma análise dos resultados da aplicação de um algoritmo genético com diferentes valores para tamanho de população, número máximo de gerações e probabilidades de mutação e recombinação em linguagens de programação diferentes. O quarto (ALANDER; MANTERE; TURUNEN, 1997) descrevia uma implementção com interesse na geração de casos de teste para testes de estresse, que levassem o software a situações extremas. O quinto (BERNDT; WATKINS, 2005) focava na geração de casos de teste para sistemas distribuídos ou em rede, gerando grandes volumes de dados ou casos de teste que levassem a intervalos de execução extensos.

Após uma leitura um pouco mais a fundo, os cinco citados anteriormente foram des-

cartados por fugirem do escopo almejado ou por terem complexidade de implementação muito alta. Dessa forma, restaram três artigos cujas premissas se destacaram: o primeiro (BOUCHACHIA, 2007) apresentando uma etapa de re-seleção para o algoritmo genético baseada em operadores de sistemas imunológicos, o segundo (DAMIA; PARVIZIMOSAED, 2021) desenvolvendo uma correção das taxas de recombinação e mutação em prol da manutenção da diversidade da população de cromossomos e o terceiro (ESNAASHARI; DAMIA, 2021) implementando novas etapas de aprendizado por reforço e de busca adicional nos cromossomos não selecionados, além da ideia de só realizar alterações em casos de teste redundantes, ou seja, os que não contribuem para a aptidão do cromossomo. Após a consideração da teoria por trás das modificações propostas, dos relatos dos códigos implementados e dos resultados dos experimentos conduzidos, foi escolhido o artigo que implementa o Memetic Algorithm for Automatic Test case generation (ESNAASHARI; DAMIA, 2021).

Como o código fonte não foi fornecido pelos autores, houve a necessidade da implementação do algoritmo MAAT seguindo o pseudocódigo apresentado e as explicações de cada etapa relatadas no artigo, mesmo com a ausência de algumas informações sobre o funcionamento do algoritmo, o que resultou no algoritmo implementado não tendo exatamente o mesmo código utilizado no artigo para os experimentos. Dessa forma, o algoritmo implementado é inspirado no MAAT original, porém não há garantia de que seu código e, consequentemente, seus resultados sejam os mesmos.

Tendo uma versão do MAAT funcional para a realização de experimentos, foi necessário considerar quatro parâmetros de inicialização que controlam a execução, com seus valores definidos podendo otimizar a performance do algoritmo e, consequentemente, seus resultados. São eles:

- limite de gerações, a quantidade máxima de iterações realizadas pelo algoritmo se ele não gerar um cromossomo com cobertura total;
- tamanho da população, a quantidade de cromossomos presentes em cada uma das gerações;
- probabilidade de mutação, a chance para cada cromossomo da população de ter o valor de um dos parâmetros alterados em um de seus casos de teste redundantes;
- limiar do aprendizado por reforço, o valor de aptidão mínimo necessário para que o melhor cromossomo participe da etapa de aprendizado por reforço.

No intuito de aumentar a amostra de experimentos possíveis para o algoritmo, foi necessária a seleção de outros códigos, além da função utilizada de exemplo no artigo do MAAT original, para terem seus casos de teste gerados pelo MAAT implementado. Como a proposta deste trabalho é a análise da eficiência do MAAT em relação a um algoritmo genético comum na geração de casos de teste de códigos de introdução à programação,

foram escolhidos para a realização dos outros experimentos alguns dos códigos utilizados na disciplina oferecida pelo Instituto de Computação para a formação de estudantes de diversos cursos da UFRJ. No Quadro 1 é possível ver os 37 códigos de introdução à programação selecionados. Desses códigos, apenas 10 seriam aceitos pelo MAAT original considerando a etapa de aprendizado por reforço descrita no artigo, sendo esses os únicos códigos em que todos os parâmetros de entrada das funções são número inteiros. Entretanto, considerando as modificações realizadas para que o algoritmo também pudesse gerar casos de teste para funções com parâmetros de entrada dos tipos string e lista de números inteiros, a quantidade total de códigos aceitos entre os selecionados aumentou para 29, 19 a mais do que se fosse utilizada a implementação original do MAAT. Ainda assim, 8 dos códigos não puderam ser utilizados nos experimentos com o MAAT implementado por possuirem parâmetros de entrada de tipos que seguiram não aceitos pelo algoritmo, como tuplas, dicionários e matrizes.

Quadro 1 – Comparação de aceitação de funções pelos algoritmos

	Função	Tipos dos parâmetros	MAAT original	MAAT implementado
1	fatorial	Inteiro	sim	sim
2	qtd_divisores	Inteiro	sim	sim
3	primo	Inteiro	sim	sim
4	soma_h	Inteiro	sim	sim
5	triangle	Inteiros	sim	sim
6	campeonato	Inteiros	sim	sim
7	avioesDePapel	Inteiros	sim	sim
8	carros	Inteiros	sim	sim
9	bolo	Inteiros	sim	sim
10	piramide	Inteiros	sim	sim
11	intercala	Listas de inteiros	não	sim
12	acima_da_media	Lista de inteiros	não	sim
13	repetidos	Lista de inteiros	não	sim
14	faltante1	Lista de inteiros	não	sim
15	multiplos	Inteiro e lista de inteiros	não	sim
16	colchao	Lista de inteiros e Inteiro	não	sim
17	insere	Lista de inteiros e Inteiro	não	sim
18	maiores	Lista de inteiros e Inteiro	não	sim
19	hashtag	String	não	sim
20	numPalavras	String	não	sim
21	conta_frase	String	não	sim
22	sempontuacao	String	não	sim
23	invertefrase	String	não	sim
24	uppCons	String	não	sim
25	freq_palavras	String	não	sim
26	lingua_p	String	não	sim
27	concatenacao	Strings	não	sim
28	substitui	Strings e Inteiro	não	sim
29	posLetra	Strings e Inteiro	não	sim
30	total	Lista de strings e Dicionário	não	não
31	filtra_pares	Tupla	não	não
32	colisao	Tuplas	não	não
33	quadrada	Matriz	não	não
34	media_matriz	Matriz	não	não
35	melhor_volta	Matriz	não	não
36	conta_numero	Inteiro e Matriz	não	não
37	busca	Inteiro e Matriz	não	não
Total			10/37	29/37

Utilizando alguns desses códigos de introdução à programação como *software under test* (SUT), o programa para qual os casos de teste são gerados, foram realizados múltiplos experimentos variando os valores das variáveis de controle de execução para encontrar os que produzem os melhores resultados em questão de três medidas:

- tempo médio de execução do algoritmo, para garantir que o tempo não passe do aceitável, o que tornaria o uso do algoritmo inviável;
- número médio de gerações necessárias para encontrar um conjunto de casos de teste que garanta a cobertura total considerando o critério de conjuntos de nós em caminhos, para garantir que a performance do algoritmo não está sendo limitada pelo limite de gerações definido;
- percentual de coberturas totais durante as execuções, ou seja, a razão entre a quantidade de execuções que obtiveram cobertura total dos requisitos de teste e o número de execuções realizadas, para garantir um percentual de vezes aceitável em que o conjunto de casos de teste gerado obteve cobertura total, sendo um resultado ideal para os testes dos códigos dos alunos.

Em outras situações de projetos de teste, uma métrica importante a ser levada em conta seria o número mínimo de casos de teste gerados necessários para satisfazerem todos os requisitos, porém como o critério de cobertura de conjuntos de nós por caminhos foi projetado de forma que cada caso de teste satisfizesse apenas um requisito, essa medida de performance não pôde ser considerada na análise dos resultados dos experimentos com o MAAT. Outra métrica possível e comumente utilizada em análises de resultados de algoritmos evolutivos, o número de chamadas de avaliação de função que o algoritmo necessita, não foi utilizada neste trabalho.

Após encontrar os valores ideais, foi possível comparar seus resultados aos produzidos por um algoritmo genético comum, o qual foi implementado a partir do MAAT, retirando as etapas de aprendizado por reforço e busca adicional e deixando de fazer a distinção entre casos de teste redundantes e não redundantes na aplicação de alterações ao cromossomo. Tanto o código fonte da implementação do MAAT quanto o do algoritmo genético comum foram disponibilizados no github[1], ambos em Python. Tendo os valores ideais para as variáveis de controle de execução do MAAT, os mesmos também foram definidos para o algoritmo genético comum, com exceção do limiar de aprendizado por reforço, cuja etapa não existe no algoritmo. Dessa forma, foi possível comparar para os mesmos SUTs o tempo médio de execução e de iterações realizadas e o percentual de cobertura total para quantificar a diferença de resultados entre os dois algoritmos.

[1] https://github.com/jhpsa/MAAT

3.2 IMPLEMENTAÇÃO DO MAAT

Assim como citado na seção anterior, os autores do artigo do MAAT não forneceram o código fonte do algoritmo, então houve a necessidade de implementá-lo seguindo o pseudocódigo e as explicações de cada etapa relatadas no artigo. Foi escolhida como SUT inicial a função triangle apresentada na subseção 2.1.2, que é a função utilizada como exemplo no artigo. O código da função foi alterado para retornar uma lista com os números que identificavam as seções do código que haviam sido executadas, sendo essas seções compostas pelos comandos que sempre executam sequencialmente, representando os nós do grafo de fluxo de controle do SUT. Dessa forma, seria possível saber o caminho da CFG percorrido pela execução do SUT com cada caso de teste. Vale ressaltar que o termo caso de teste possui um significado mais abrangente, assim como explicitado na subseção 2.1.1, porém, para seguir a terminologia presente no artigo da implementação do MAAT original, neste trabalho o termo será utilizado para descrever o conjunto de valores atribuídos aos parâmetros de entrada da função SUT em uma execução na tentativa de satisfazer um dos requisitos de teste. O critério de cobertura por grafos considerado para a avaliação de aptidão dos cromossomos, o qual não foi especificado no artigo, foi escolhido como cobertura por conjuntos de nós em caminhos, sendo os requisitos de teste os conjuntos dos nós, sem repetições de nós, presentes em cada um dos caminhos possíveis da CFG.

Figura 5 – Pseudocódigo do algoritmo MAAT

Algorithm 1 The proposed MAAT algorithm.

```
1: inputs
     The SUT,
     population size,
     max_iterations,
     memtic_threshold
2: output
     Test set for the SUT
3:   Create CFG of the SUT and obtain its paths
4:   Population = [ ]
5:   iter = 1
6:   for i = 1 to PopulationSize do
7:     t_i = RandomChromosome ()
8:     Population[i] = t_i
9:   while (not all paths are covered) and (iter < max_iterations)
10:    EvaluatePopulation ()
11:    Recombination ()
12:    Mutation ()
13:    if f_best >= memtic_threshold:
14:      MemticPreprocess()
15:      q-learning()
16:    AdditionalSearch ()
17:    Population = Selection ()
18:    iter++
19: return t_best
```

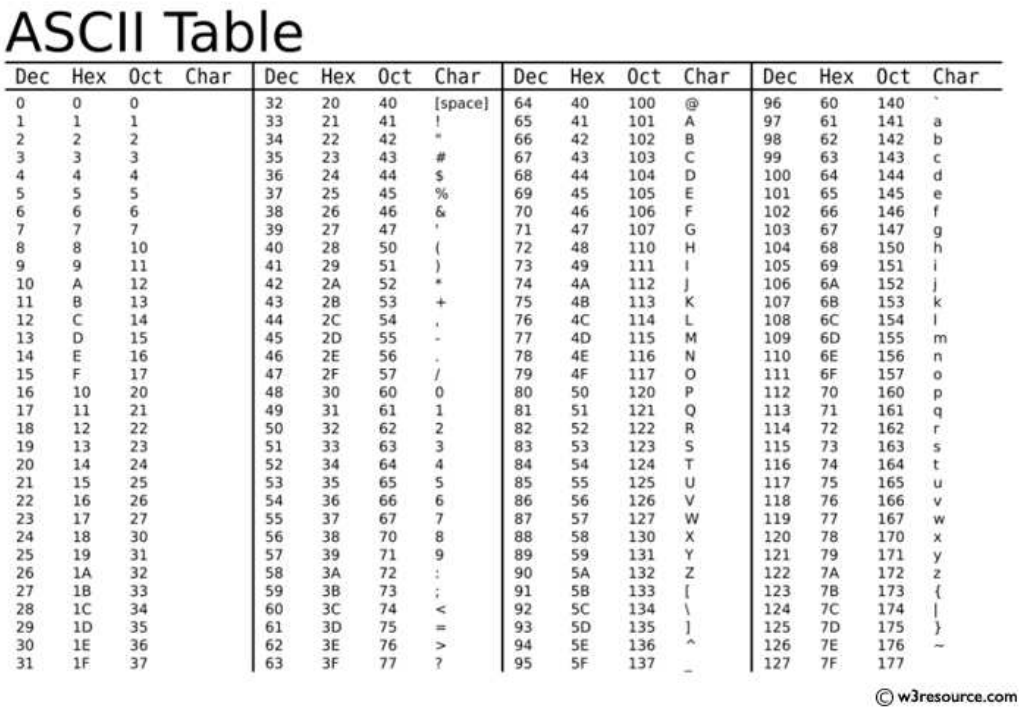
Fonte: Esnaashari, Mehdi e Damia, Amir Hossein (2021, p. 8)

O algoritmo implementado começa com a definição de alguns parâmetros para controle, como qual o SUT, o tamanho da população, o número máximo de iterações, a probabilidade de mutação, que não havia no original, e o limiar da etapa de aprendizado por reforço. Posteriormente, ocorre a criação da população inicial de cromossomos. Para isso, é criada uma lista com uma quantidade de cromossomos igual ao tamanho predefinido da população. Cada cromossomo é uma lista com quantidade de elementos igual ao número de requisitos de teste, sendo cada elemento um caso de teste, uma possibilidade de solução para satisfazer um requisito de teste. Vale ressaltar que em outros projetos de teste seria possível que um caso de teste satisfizesse mais de um requisito, porém o MAAT foi desenvolvido de forma a gerar casos de teste que só satisfazem um requisito. Cada caso de teste é uma lista de tamanho igual ao número de parâmetros de entrada recebidos pelo SUT. No algoritmo original, essa etapa foi projetada para só gerar casos de teste com valores para parâmetros inteiros. Para poder utilizar um número maior de SUTs diferentes nos experimentos, foram realizadas modificações para gerar casos de testes para funções com parâmetros dos tipos lista de inteiros e strings também. O tipo dos parâmetros esperados pelo SUT é checado, só aceitando inteiros, listas de inteiros ou strings, e para cada parâmetro é gerado aleatoriamente um elemento do mesmo tipo na lista do caso de teste, de forma que:

- se o parâmetro for do tipo inteiro, é gerado aleatoriamente um inteiro entre 0 e 100;
- se for uma lista, é criada uma lista com tamanho gerado aleatoriamente entre 0 e 10 e cada elemento da lista será um inteiro gerado aleatoriamente entre 0 e 100;
- se for uma string, são gerados um número aleatório entre 0 e 10 de caracteres aleatórios dentre os com identificador decimal 32 até 137 da tabela ASCII que são concatenados em uma string.

A escolha do domínio de valores possíveis a serem gerados aleatoriamente para cada um do tipos de parâmetro foi determinada de maneira arbitrária. Trabalhos futuros devem explorar se outras escolhas seriam melhores.

Como a geração automatizada de casos de testes é um problema de otimização, esses valores são considerados os limites mínimos e máximos do problema para cada tipo de parâmetro, e os mesmos não foram especificados no artigo da implementação original.



ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Figura 6 – Tabela ASCII

Tendo a população inicial de cromossomos criada, a parte principal do algoritmo acontece dentro de um laço que repetirá enquanto ainda houver algum requisito de teste não satisfeito, ou seja, um conjunto de nós de um caminho da CFG não coberto pelo cromossomo melhor avaliado da população atual, e não houverem sido realizadas o número máximo de iterações. Ou seja, no momento em que uma das condições de parada forem alcançadas, no caso em que o melhor cromossomo atingir a cobertura total ou o máximo de iterações forem executadas, o laço é concluído.

Dentro do laço principal, a primeira etapa é a de avaliação da aptidão de cada cromossomo da população atual, exibida na Figura 7. Para isso, é checado quais nós da CFG cada caso de teste do cromossomo visita na execução do SUT, e cada conjunto desses é armazenado em uma lista. Essa lista então é comparada com a de conjuntos de nós dos caminhos da CFG, os requisitos de teste. A aptidão de cada cromossomo será a razão entre quantos conjuntos de nós dos caminhos da CFG estão na sua lista e o número total de conjuntos de nós em caminhos da CFG, ou seja, o percentual de cobertura do critério de conjuntos de nós em caminhos. Considere C o cromossomo sendo avaliado, *F(C)* como o valor de aptidão desse cromossomo, *CFG* o grafo de fluxo de controle gerado a partir da SUT. Sendo *L(C)* a lista dos conjuntos de nós da CFG visitados na execução de cada caso de teste do cromossomo pela SUT excluindo possíveis repetições e *L(CFG)* a lista dos conjuntos de nós em caminhos da CFG, ou seja, os requisitos que queremos atender:

$$F(C) = \frac{|L(C)|}{|L(CFG)|}$$

Outra forma de pensar nesse valor de aptidão do cromossomo seria também a razão entre a quantidade de requisitos satisfeitos pelo conjunto de casos de teste do cromossomo ($|RS(C)|$), ou seu número de casos de teste não redundantes ($|NR(C)|$), que são equivalentes a $|L(C)|$ e a quantidade total de requisitos de teste projetados para a SUT ($|R(SUT)|$), ou o número de casos de teste de cada cromossomo ($|CT(C)|$), que são equivalentes a $|L(CFG)|$.

$$A = |RS(C)| = |NR(C)|$$

$$B = |R(SUT)| = |CT(C)|$$

$$F(C) = \frac{A}{B}$$

Após a avaliação de todos os cromossomos, a população é reordenada de forma crescente baseada no valor de aptidão de cada cromossomo.

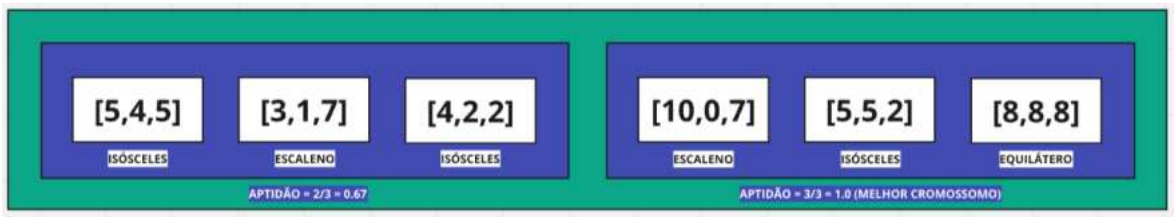


Figura 7 – Aptidão de possíveis cromossomos para a função triangle

A segunda etapa do laço principal é a de recombinação, onde dois cromossomos são escolhidos aleatoriamente para conceberem dois novos cromossomos que irão substituí-los na população, como ilustrado na Figura 8. O processo será executado uma quantidade de vezes aleatória, já que o número de recombinações não foi explicitado no artigo da implementação original. Poderão ser selecionados de dois à quantidade total de cromossomos para formarem pares, de forma que nunca serão selecionados de novo cromossomos já recombinados anteriormente. Para a geração dos novos cromossomos é necessário, para cada um dos dois cromossomos pais, separar os casos de teste em dois conjuntos, um com os casos de teste redundantes, ou seja, casos de teste que executem no SUT um caminho da CFG que já esteja sendo coberto por outro caso de teste do mesmo cromossomo, e outro com os não redundantes. Entre dois ou mais casos de teste do cromossomo que sejam redundantes entre si, o primeiro da lista será considerado não redundante e o resto, redundante. Dessa forma, um caso de teste aleatório não redundante de um dos cromossomos pais substitui um caso de teste aleatório redundante do outro cromossomo e vice-versa, assim produzindo os dois novos cromossomos da população. Vale ressaltar que o caso de teste não redundante de um cromossomo pode ser redundante no outro, então não há uma garantia de evolução em toda iteração da etapa de recombinação, porém como o caso de teste a ser substituído é redundante, no pior dos casos a aptidão do cromossomo seguirá

a mesma, então considerando diversas iterações, a tendência é da melhora da aptidão da população afetada pela recombinação.

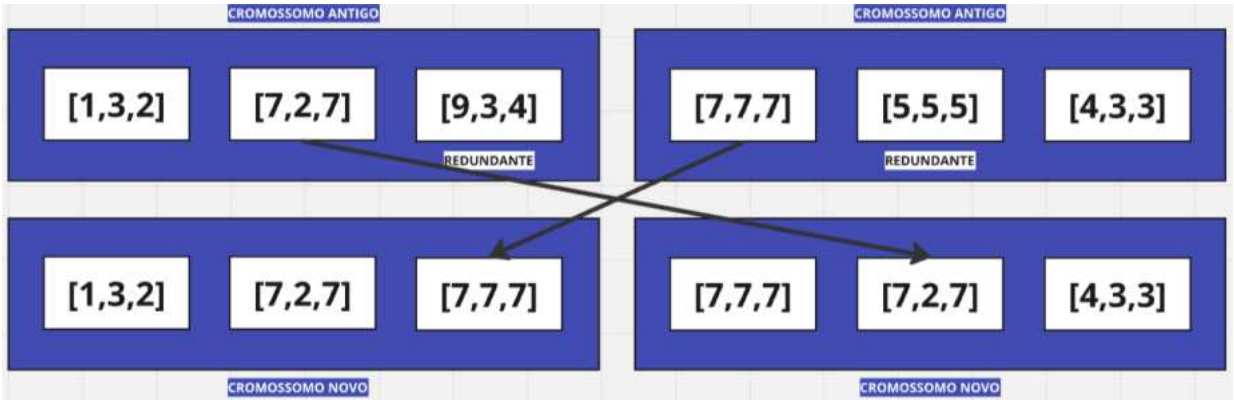


Figura 8 – Recombinação de possíveis cromossomos para a função triangle

Na terceira etapa, a mutação, todos os cromossomos terão uma probabilidade predefinida para que um de seus casos de teste redundantes sofra uma alteração aleatória no valor de um dos parâmetros, como mostrado na Figura 9. Um processo para selecionar o caso de teste redundante a sofrer mutação foi citado no artigo da implementação original, porém não explicado e, após pesquisa sobre tal processo, foi considerado complexo demais para implementação. O caso de teste redundante será escolhido aleatoriamente para a mutação. A mutação realizada será a troca do parâmetro escolhido aleatoriamente no caso de teste redundante por outro parâmetro do mesmo tipo criado da mesma forma que quando um parâmetro desse tipo é criado em um caso de teste de um cromossomo na criação da população inicial. Como apenas os casos de teste redundantes poderão ser alterados, pela mesma lógica da etapa de recombinação, a aptidão de um cromossomo nunca diminuirá e, consequentemente, a tendência a cada iteração será de um aumento da aptidão dos cromossomos afetados pela mutação.

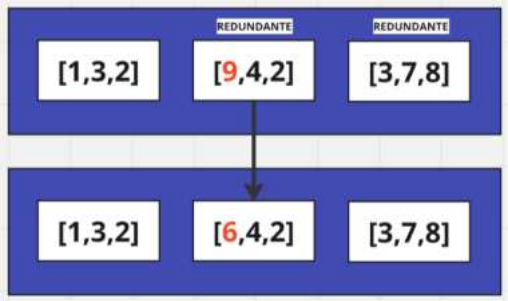


Figura 9 – Mutação de um possível cromossomo para a função triangle

A quarta etapa, demonstrada na Figura 10, tem como objetivo tentar melhorar a aptidão do melhor cromossomo, de forma que os cromossomos da população são reavalidados

e reordenados após as etapas de recombinação e mutação. No MAAT original, seria utilizado o cromossomo melhor avaliado inicialmente, porém isso gerou erros. Essa etapa só será realizada se esse cromossomo já não satisfizer todos os requisitos de teste, pois nesse caso seria impossível aumentar sua aptidão, mas também não ocorrerá se a aptidão desse cromossomo não se equiparar ou ultrapassar o valor do limiar predefinido. Atendendo essas duas condições, serão procurados todos os requisitos de teste ainda não satisfeitos por nenhum dos casos de teste do cromossomo e um desses requisitos será escolhido aleatoriamente. Tendo o conjunto de nós do requisito almejado escolhido, será calculado, entre todos os casos de teste do cromossomo, aquele cujo conjunto de nós resultante da execução do SUT for o mais parecido com o que se deseja cobrir. A similaridade entre os conjuntos será calculada pelo índice de Jaccard, sendo este o resultado da razão entre a interseção entre os conjuntos de nós do caminho analisado e o desejado e a união desses conjuntos. Dessa forma, sendo J o índice de Jaccard e A e B conjuntos:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

O caso de teste cujo conjunto de nós do caminho na CFG possuir o maior índice de Jaccard em relação ao caminho desejado será o selecionado para participar do aprendizado por reforço Q-learning. O processo então será modelado matematicamente com uma representação em espaço de estados. O número de estados, k, será igual ao número de parâmetros de entrada da SUT, de forma que o espaço de estados será definido como:

$$S = \{s(1), s(2), ...s(k)\}$$

Já o caso de teste atual, que possui um elemento para cada parâmetro de entrada da SUT, será representado por:

$$t_{cur} = [x_{cur}(1), x_{cur}(2), ...x_{cur}(k)]$$

Em cada estado, será definida para ser realizada uma ação dentre um conjunto de possíveis ações para alterar o valor *x(l)* correspondente ao estado *s(l)*. Existem oito possibilidades de ações que poderão ser realizadas para alterar o parâmetro do estado, considerando o tipo do parâmetro do estado e do parâmetro do estado seguinte ou anterior. Todas as alterações envolvendo parâmetros do tipo lista de inteiros e string tiveram que ser decididas tentando encontrar alguma equivalência com as alterações para inteiros detalhadas no artigo da implementação original. As possibilidades são:

1. subtrair um de um dos inteiros (lista de inteiros), substituir um dos caracteres pelo caractere anterior (string) ou subtrair um do valor do parâmetro (inteiro);

2. incrementar um a um dos inteiros (lista de inteiros), substituir um dos caracteres pelo caractere seguinte (string) ou incrementar um ao valor do parâmetro (int);

3. inserir um caractere (string) ou não fazer nada (lista de inteiros e inteiro);

4. copiar o valor de um dos inteiros para o da posição anterior (lista de inteiros), copiar um dos caracteres para o da posição anterior (string), copiar o valor de um dos inteiros da lista (inteiro seguido por lista de inteiros), copiar o valor numérico do código unicode de um dos caracteres da string (inteiro seguido por string) ou copiar o valor do parâmetro seguinte (inteiro seguido por inteiro);

5. copiar o valor de um dos inteiros para o da posição seguinte (lista de inteiros), copiar um dos caracteres para o da posição seguinte (string), copiar o valor de um dos inteiros da lista (inteiro seguinte a lista de inteiros), copiar o valor numérico do código unicode de um dos caracteres da string (inteiro seguinte a string) ou copiar o valor do parâmetro anteior (inteiro seguinte a inteiro);

6. esvaziar a lista (lista de inteiros), deletar um dos caracteres (string) ou alterar o valor do parâmetro para zero (inteiro);

7. trocar o valor de um dos inteiros com o da posição anterior (lista de inteiros), trocar um dos caracteres com o da posição anterior (string), trocar o valor com um dos inteiros da lista (inteiro seguido por lista de inteiros), trocar o valor do inteiro pelo valor numérico do código unicode de um dos caracteres da string e o caractere pelo que tiver o código unicode do valor do parâmetro (inteiro seguido por string) ou trocar o valor com o do parâmetro seguinte (inteiro seguido por inteiro);

8. trocar o valor de um dos inteiros com o da posição seguinte (lista de inteiros), trocar um dos caracteres com o da posição seguinte (string), trocar o valor com um dos inteiros da lista (inteiro seguinte a lista de inteiros), trocar o valor do inteiro pelo valor numérico do código unicode de um dos caracteres da string e o caractere pelo que tiver o código unicode do valor do parâmetro (inteiro seguinte a string) ou trocar o valor com o do parâmetro anterior (inteiro seguinte a inteiro).

No caso do valor do parâmetro do estado atual, anterior ou seguinte ser uma string vazia, as opções 1, 2 e 5 não farão nada enquanto as outras concatenarão um caractere aleatório. A primeira iteração sempre estará no primeiro estado, então sua alteração será realizada no primeiro valor do caso de teste. Após a alteração ser realizada, o caso de teste modificado será:

$$t_{next} = [x_{next}(1), x_{next}(2), ...x_{next}(k)]$$

Dessa forma, a SUT será executada com os valores do caso de teste modificado como parâmetros. Dado o conjunto de nós visitados no caminho dessa execução, o índice de Jaccard será calculado e comparado com o índice do caso de teste atual. O caso de teste com maior valor do índice de Jaccard será utilizado como caso de teste atual no estado da

iteração seguinte. O estado seguinte será escolhido aleatoriamente. Cada estado tem uma probabilidade igual de transicionar para cada outro estado, inclusive si mesmo. Assim, o estado seginte será escolhido com probabilidade 1/k. Um laço será repetido enquanto o caso de teste resultante não cobrir o requisito desejado e um número máximo de iterações definido ainda não tiver sido atingido, ou seja, no momento em que o requisito for coberto ou que já tiverem sido realizadas o número máximo de repetições, o laço do Q-learning chegará ao fim. Dentro do laço, cada iteração realizará a escolha da opção de alteração, a alteração, a checagem da cobertura do requisito e, se o requisito não tiver sido coberto, a comparação dos índices de Jaccard e a escolha do estado seguinte. Dessa forma, a recompensa de cada iteração do aprendizado por reforço é a possibilidade de gerar um conjunto de casos de teste que cubra um caminho ainda não coberto, satisfazendo mais um requisito e aumentando o percentual de cobertura do cromossomo. Quando o laço chegar a seu fim, tendo coberto o conjunto de nós do caminho desejado ou não, o caso de teste resultante substituirá um dos casos de teste redundantes escolhido aleatoriamente no cromossomo.

Figura 10 – Q-learning de um possível cromossomo para a função triangle

A quinta e última etapa, que ocorrerá independente da etapa anterior ter sido realizada ou não, será a de seleção e busca adicional, como mostrado na Figura 11. Primeiramente, serão selecionados, através de uma seleção baseada na aptidão inicial, os cromossomos da população atual que sobreviverão para a próxima geração, ou seja, farão parte da população inicial da próxima iteração do algoritmo. Um método específico de seleção por ranqueamento foi citado no artigo da implementação original, porém não explicado e, após pesquisa sobre tal método, foi considerado complexo demais para implementação. O caso de teste redundante será escolhido aleatoriamente para a mutação. Para cada cromossomo, a probabilidade da sua seleção para a próxima geração será igual à razão entre sua posição na população (ordenada de forma crescente baseada na aptidão inicial) e o tamanho da população, ou seja, cromossomos com maior aptidão terão maior probabilidade de seleção, mas a diferença de probabilidade de seleção entre dois cromossomos será uniforme, e não proporcional à diferença de suas aptidões. Após a seleção, será realizada uma busca para checar se existe algum caso de teste entre os cromossomos não selecionados que cubra um conjunto de nós de um caminho da CFG que não esteja sendo coberto por nenhum dos

casos de teste dos cromossomos selecionados e, se houver um ou mais casos de teste nessa situação, os mesmos substituirão casos de teste redundantes aleatoriamente escolhidos no cromossomo com a maior aptidão inicial entre os selecionados.

Figura 11 – Seleção e busca adicional de uma possível população para a função triangle

Finalmente, tendo o conjunto de cromossomos da população atual selecionados para participarem da geração seguinte, para preencher o resto da população dessa geração subsequente, serão gerados, da mesma forma que os da população da primeira geração, uma quantidade de novos cromossomos igual ao número de cromossomos não selecionados da geração atual. Dessa forma, o conjunto dos cromossomos antigos selecionados e dos novos gerados compõem a população inicial da próxima iteração do laço principal do algoritmo. Após a última repetição, seja tendo gerado um cromossomo que cobre todos os requisitos de teste ou por ter chegado ao número máximo de iterações, o conjunto de casos de testes resultante do algoritmo MAAT será o do cromossomo de maior aptidão da última geração.

3.3 EXEMPLO DE UTILIZAÇÃO DO MAAT

Após entender como o MAAT pode ser implementado, é necessário conhecer melhor o processo de utilização dele para geração de casos de teste. Nesse exemplo, será mostrado onde o algoritmo pode se encaixar no projeto de testes do Machine Teaching. Nessa ferramenta de ensino de introdução à programação, o processo se inicia com a publicação do enunciado de um problema de lógica, que terá associado a ele um código solução, este último não disponível para os alunos, como exemplificados a seguir.

Problema: Faça uma função chamada **primo** que dado um número inteiro positivo, verifique se este número é primo ou não. Retorne um valor booleano. *Dica:* uma estratégia simples para identificar a primalidade de um número é verificar se não existe nenhum número menor que ele próprio (e maior ou igual a 2) que o divida. *Dica 2*: O número de divisões indicado na dica anterior é maior que o necessário. Você consegue reduzi-lo?

Código solução:

```
def primo(n: int):
    if n<=1:
        return False
    elif n==2:
        return True
    for contador in range(2,n):
        if n%contador == 0:
            return False
    return True
```

Figura 12 – Código da função primo

Com a implementação do MAAT sem integração com uma ferramenta de geração automática de CFGs, é necessário alterar o código solução para que a função retorne uma lista com os números associados aos nós da parte do código que é acessada em uma execução. A seguir é mostrada uma possibilidade de alteração da função original para ser utilizada no MAAT.

Código alterado manualmente:

```
def primo(n: int):
    path =[]
    if n<=1:
        path.append(1)
        #return False
        return path
    elif n==2:
        path.append(2)
        #return True
        return path
    for contador in range(2,n):
        if n%contador == 0:
            path.append(3)
            #return False
            return path
    path.append(4)
    #return True
    return path
```

Figura 13 – Código alterado da função primo

Os nós associados a números devem ser incluídos em partes do código de forma a diferenciar os possíveis caminhos de execução. O exemplo acima demonstra uma possibilidade de implementação com o menor número possível de nós numerados para esse caso,

de forma que todo caminho de execução visite pelo menos um nó numerado, para que não haja a possibilidade de existir um conjunto de nós vazio resultante de uma execução, o que geraria problemas na forma como o MAAT foi implementado. Também é necessário substituir os retornos originais da função e colocar para que seja retornada no mesmo ponto a lista de nós.

Dada essa alteração no código com apenas quatro nós numerados, a CFG da função pode ser modelada como a seguir. Vale ressaltar que essa CFG, que poderia ser modelada de outras formas, possui mais de 4 nós, sendo o número mínimo de nós do modelo ditado por suas estruturas condicionais e de repetição. Os nós numerados representam os explicitados na alteração do código.

Possível CFG:

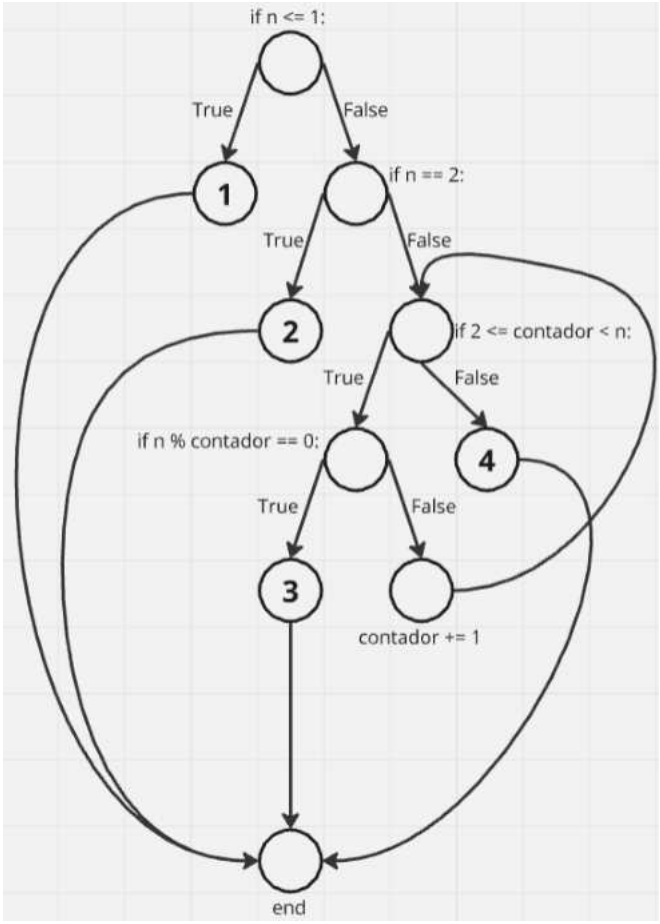


Figura 14 – CFG da função primo

O modelo auxilia no entendimento dos caminhos possíveis da CFG, mas não é obrigatório para o processo. O conjunto de requisitos de teste, exibido a seguir, é definido na inclusão dos nós numerados na alteração do código. Vale lembrar que a criação desses requisitos para serem utilizados no MAAT seguem o critério de cobertura de conjuntos de nós em caminhos, definido na subseção 2.1.2.

Conjunto de requisitos de teste: [[1], [2], [3], [4]]

Finalmente, após encontrar o conjunto de requisitos de teste do critério de cobertura por conjuntos de nós em caminhos do código solução alterado do problema publicado na Machine Teaching, é possível utilizar o MAAT com o código alterado da função SUT e o conjunto de requisitos de teste dele para gerar um conjunto de casos de teste com percentual de cobertura desejado para testar os códigos dos alunos submetidos para o problema. Ao realizar uma execução do MAAT implementado para o cenário desse exemplo, o conjunto de casos de teste gerado e os dados da execução do MAAT foram esses a seguir.

Possível conjunto de casos de teste gerado: [[65], [2], [0], [59]]

Dados da execução: 2 iterações, 4 milissegundos e 100% de cobertura

3.4 INTEGRAÇÃO COM MACHINE TEACHING

Atualmente, para publicar um problema na Machine Teaching é necessário preencher um formulário com informações do problema como título, enunciado e dificuldade como ilustrado na Figura 15. Além dessas informações, também é exigido o código de uma função geradora de casos de teste para criar o conjunto de valores de entrada que serão utilizados para testar os códigos dos alunos com a tentativa de resolução do problema.

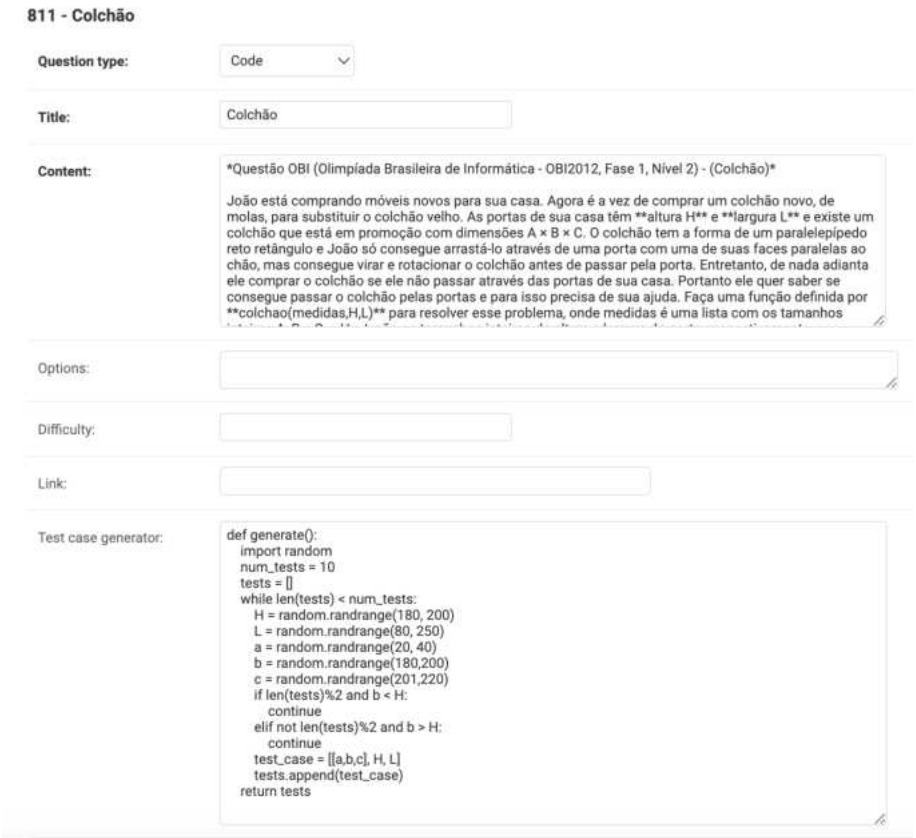


Figura 15 – Formulário de publicação de problema na Machine Teaching

Em seu estado atual, é possível utilizar o algoritmo MAAT implementado como função geradora de um problema da Machine Teaching, entretanto será necessário incluir também nesse campo do formulário o código solução do problema alterado e seu conjunto de requisitos de teste, como demonstrado nos exemplos da seção 3.3. Idealmente, feita como trabalho futuro uma integração do MAAT implementado com uma ferramenta de geração de CFGs, só seria necessário o código solução junto ao MAAT. Além disso, o MAAT implementado também possui uma limitação quanto a tipos de parâmetros de entrada da função testada, então até que fosse implementada uma solução para tal, o MAAT só poderia ser utilizado para problemas da Machine Teaching cujos parâmetros esperados fossem inteiros, listas de inteiros e strings. Dessa forma, poderia ser implementado nesse formulário de publicação de problema da Machine Teaching a opção de, se os parâmetros de entrada esperados fossem aceitos pelo algoritmo, escolher utilizar o MAAT em vez de ter que definir uma função geradora de casos de teste.

4 EXECUÇÃO DO EXPERIMENTO

Neste capítulo, serão exibidos os experimentos descritos na seção 3.1 para que seus resultados possam ser analisados. Dessa forma, na seção 4.1 serão descritos os experimentos do MAAT implementado onde o algoritmo foi testado para uma mesma função alterando uma das variáveis de controle de execução com o objetivo de encontrar o valor de cada variável que otimize a execução do algoritmo. Os dados completos desses experimentos podem ser encontrados nos apêndices. Tendo encontrado os valores ideais para esses parâmetros, na seção 4.2 serão detalhados os experimentos comparando os resultados de execução do MAAT implementado com os resultados do algoritmo genético comum derivado dele, os dois algoritmos utilizando os valores definidos nos experimentos anteriores para os parâmetros de controle de execução. Vale notar que cada experimento individual, cujos resultados são apresentados em uma linha da tabela, é realizado executando o algoritmo 1000 vezes em sequência para garantir que as médias calculadas tendam a ser representações fiéis da realidade.

4.1 ENCONTRANDO OS VALORES IDEAIS PARA OS PARÂMETROS DO MAAT

No capítulo anterior, foram apresentadas as quatro variáveis de controle de execução cujos valores influenciam na eficiência da execução do MAAT implementado, sendo elas o limite de gerações, o tamanho da população, a probabilidade de mutação e o limiar do aprendizado por reforço. Para definir os valores dessas variáveis de forma a otimizar os resultados do algoritmo, foram realizados experimentos utilizando o MAAT para gerar casos de teste automatizados para alguns dos códigos do conjunto de funções aceitas por ele listadas no Quadro 1. A lista das oito funções que foram utilizadas como SUT nesses experimentos, escolhidas pela sua maior complexidade de caminhos e laços além dos diferentes tipos de seus parâmetros de entrada, pode ser encontrada no Quadro 6. Muitas das funções aceitas pelo MAAT implementado não foram selecionadas para participarem dos experimentos por terem estruturas simples demais, sem laços ou bifurcações no caminho da CFG, o que se é esperado considerando que pela natureza deste trabalho os códigos de interesse pertencem ao escopo da introdução à programação. Se todas as funções fossem utilizadas nos experimentos, a maioria delas tenderia a ter seus casos de teste ideais encontrados de forma imediata utilizando quaisquer valores para as variáveis de controle de execução, portanto resultando em uma grande quantidade de dados inconclusivos ou até mesmo tendenciosos para valores insuficientes para as funções mais complexas. Logo, a escolha de filtrar o conjunto de forma a selecionar apenas essas oito funções prejudica o tamanho da amostra de códigos a serem utilizados como SUT porém diminui o risco de comprometer as conclusões formadas em cima dos dados resultantes dos experimentos.

Foram realizados quatro experimentos, cada um com o objetivo de analisar os efeitos da alteração do valor de uma das variáveis de controle de execução, ou seja, de forma a comparar os resultados da execução do algoritmo nas mesmas exatas situações, apenas com uma das variáveis sendo alterada. Para cada experimento, foram realizados oito conjuntos de testes, cada um utilizando uma das oito funções selecionadas como SUT e definindo valores fixos para as outras três variáveis de controle de execução além da variável sendo analisada. Esses conjuntos contém quatro testes com valores diferentes definidos para a variável em análise e com os dados de cada um resultantes de um total de 1000 execuções do MAAT implementado utilizando. Os dados registrados para cada um dos testes dos conjuntos de cada experimento foram:

a) o número máximo de gerações, ou seja, dentre as 1000 execuções, o número da última geração da execução com mais iterações;

b) média de gerações, o número médio de iterações realizadas pelo algoritmo por execução;

c) tempo médio, a quantidade média da duração, em milissegundos, da execução do algoritmo, desde a criação da sua primeira geração da população de cromossomos até o final da sua última iteração;

d) cobertura média, o percentual médio de cobertura do critério de nós em caminhos da CFG do SUT;

e) percentual de cobertura total, a razão entre o número de execuções que atingiram a cobertura total do critério e o número total de execuções.

O primeiro experimento, cujas informações completas podem ser encontradas no Apêndice A, foi realizado para descobrir o valor ideal do limite de gerações, ou seja, o número máximo de iterações que o algoritmo pode realizar enquanto não tiver alcançado cobertura total do critério de nós nos caminhos da CFG do SUT. Se o valor dessa variável for muito pequeno, o algoritmo pode ficar limitado, não podendo executar iterações suficientes para chegar a um percentual de cobertura desejável, porém se o valor definido para essa variável for grande demais o tempo de execução pode ficar prejudicado com o algoritmo realizando iterações excessivas para ganhar um aumento insignificante de percentual de cobertura em uma situação onde o percentual com menos iterações já atingiria um percentual desejável.

Para esse experimento, foram escolhidos para serem testados como limite de gerações os valores 1, 10, 50 e 100. Os valores definidos para as outras três variáveis de controle de execução foram:

a) tamanho da população = 10

b) probabilidade de mutação = 100%

c) limiar do aprendizado por reforço = 0

O tamanho da população foi escolhida inicialmente como 10 para ter um número de cromossomos suficiente para não limitar o potencial do algoritmo e os valores de probabilidade de mutação e limiar do aprendizado por reforço foram definidos de forma a garantir os melhores percentuais de cobertura possíveis para suas situações, já que a etapa de mutação e aprendizado por reforço só alteram casos de teste redundantes, então a aptidão dos cromossomos afetados nunca irá piorar. Os experimentos seguintes relatam o tamanho ideal da população e a necessidade ou não da alteração das outras duas variáveis.

Quadro 2 – Alterando o limite de gerações para a função freq_palavras

Limite de gerações	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
1	1	1	3	48,53%	0%
10	8,5	10	14	75,60%	32,90%
50	17,9	50	28	98%	93,50%
100	18,3	100	28	99,93%	99,80%

Por mais que em alguns casos, para as funções mais simples, o número máximo de gerações não tenha ultrapassado 10, nos conjuntos de testes das funções mais complexas, como a freq_palavras exibida no Quadro 2, mesmo o limite de 100 gerações limitou de certa forma a execução do algoritmo, porém na maioria dos casos a cobertura média e percentual de cobertura total não aumentaram tanto entre os limites de 50 e 100 ou no limite de 100 as coberturas se aproximaram do 100%. Dessa forma, o valor da variável de limite de gerações utilizado no restante do testes foi de 100, com o intuito de limitar menos o potencial de cobertura do algoritmo já que o aumento do tempo médio, por ser apenas milissegundos, foi desconsiderado na escolha, porém se fosse levado em conta a razão de cobertura e tempo, um limite mais próximo de 50 seria uma melhor escolha.

O segundo experimento, cujas informações completas podem ser encontradas no Apêndice B, foi realizado para descobrir o valor ideal do tamanho da população, ou seja, o número de cromossomos existentes em cada geração. Se o valor dessa variável for muito pequeno, haverá menores chances de que um cromossomo com alta aptidão seja gerado nas etapas inicial e de mutação e o baixo número de cromossomos conterá menor diversidade, o que dificulta a eficiência da etapa de recombinação, porém se o valor definido para essa variável for grande demais a execução de cada iteração do algoritmo pode acabar sendo desnecessariamente demorada por ter que processar todas as ações em um número de cromossomos onde muitos não agregam mais diversidade ao processo.

Para esse experimento, foram escolhidos para serem testados como tamanho da população os valores 1, 10, 50 e 100. Os valores definidos para as outras três variáveis de controle de execução foram:

a) limite de gerações = 100

b) probabilidade de mutação = 100%

c) limiar do aprendizado por reforço = 0

O limite de gerações foi escolhido como 100 de acordo com os resultados do experimento anterior e os valores da probabilidade de mutação e limiar do aprendizado por reforço se mantiveram pelos motivos citados anteriormente.

Quadro 3 – Alterando o tamanho da população para a função multiplos

Tamanho da população	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
1	70,5	100	8	76,95%	51,80%
10	42,7	100	72	94,18%	85,70%
50	40,5	100	354	95,15%	87%
100	42,3	100	722	94,60%	86,90%

Na maioria dos conjuntos de testes, o valor de tamanho de população que teve maior aumento nos percentuais de cobertura em relação ao valor testado anteriormente foi 10, como na função multiplos apresentada no Quadro 3. Em alguns poucos casos esse valor foi 1, porém apenas para as funções mais simples. Executar o algoritmo para uma função mais complexa com população de apenas um cromossomo limitaria seu potencial. Em nenhum dos conjuntos de testes os valores de 50 ou 100 apresentaram grande aumento de eficiência em relação ao teste com população 10, então o valor da variável de tamanho da população continuou definido como 10 para o restante dos experimentos.

O terceiro experimento, cujas informações completas podem ser encontradas no Apêndice C, foi realizado para descobrir o valor ideal da probabilidade de mutação, ou seja, a chance calculada para cada cromossomo da população ter um de seus casos de teste redundantes alterado. Se o valor dessa variável for muito pequeno, haverá menores chances de mutação, diminuindo a possibilidade de que um caso de teste redundante deixe de ser e aumente a aptidão do cromossomo, porém se o valor definido para essa variável for muito alto pode haver mutações desnecessárias em cromossomos demais, fazendo com que o tempo de execução aumente sem uma contribuição equivalente.

Para esse experimento, foram escolhidos para serem testados como probabilidade de mutação os valores 10%, 35%, 65% e 90%. Os valores definidos para as outras três variáveis de controle de execução foram:

a) limite de gerações = 100

b) tamanho da população = 10

c) limiar do aprendizado por reforço = 0

O limite de gerações foi escolhido como 100 e o tamanho da população como 10 de acordo com os resultados dos experimento anteriores e o valor do limiar do aprendizado por reforço se manteve pelos motivos citados anteriormente.

Quadro 4 – Alterando a probabilidade de mutação para a função faltante1

Probabilidade de mutação	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
10%	97,7	100	74	74,28%	5%
35%	95,5	100	75	75,53%	9%
65%	93,84	100	76	76,40%	12,20%
90%	90,7	100	79	77,43%	18,40%

Muitos dos conjuntos de teste foram inconclusivos, pois os dados coletados resultantes dos testes realizados não apresentaram diferenças significativas entre si. Entretanto, os conjuntos de teste que foram conclusivos indicaram que os valores mais altos de probabilidade de mutação obtiveram resultados melhores sem comprometer o tempo médio de execução, como o da função faltante1 mostrada no Quadro 4. Portanto, como para algumas funções o valor não fez diferença e para os que fez, o aumento do valor melhorou alguns pontos sem prejudicar outros, o valor da variável de probabilidade de mutação se manteve em 100% no restante dos experimentos.

O quarto experimento, cujas informações completas podem ser encontradas no Apêndice D, foi realizado para descobrir o valor ideal do limiar de aprendizado por reforço, ou seja, o valor mínimo de aptidão para que o cromossomo mais bem avaliado da geração participe da etapa de aprendizado por reforço. Se o valor dessa variável for grande demais, é possível que cromossomos que se beneficiariam do Q-learning para gerar um caso de teste improvável não tenham essa oportunidade, porém se o valor definido para essa variável for muito pequeno, cromossomos com muitos casos de teste redundantes que poderiam deixar de ser redundantes de forma menos custosa através de outras etapas do algoritmo, podem acabar participando da etapa de aprendizado por reforço desnecessariamente, chegando ao mesmo resultado com um tempo de execução maior.

Para esse experimento, foram escolhidos para serem testados como limiar do aprendizado por reforço os valores 0.1, 0.35, 0.65 e 0.9. Os valores definidos para as outras três variáveis de controle de execução foram:

a) limite de gerações = 100

b) tamanho da população = 10

c) probabilidade de mutação = 100%

O limite de gerações foi escolhido como 100, o tamanho da população como 10 e a probabilidade de mutação como 100% de acordo com os resultados dos experimento anteriores.

Quadro 5 – Alterando o limiar do aprendizado por reforço para a função triangle

Limiar do aprendizado por reforço	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
0,9	99,2	100	213	62,83%	1,70%
0,65	22,4	100	51	99,13%	98,60%
0,35	22,8	100	51	99%	98,50%
0,1	1,9	3	9	100%	100%

Este experimento teve os resultados mais variados entre os quatro, com todos os valores testados tendo sido os ideais, ou seja, com aumento dos percentuais de cobertura mais significativos em relação ao valor anterior, para alguma das funções testadas. Para a função triangle por exemplo, encontrada no Quadro 5, a diminuição do limiar do aprendizado por reforço de 0,9 para 0,65 fez com que o percentual de cobertura total pulasse de 1,7% para 98,6% mas só chegou na cobertura total para todos os casos com o limiar em 0,1. Entretanto, de forma semelhante à situação do experimento da análise da probabilidade de mutação, por mais que algumas funções mais simples tenham obtido resultados decentes mesmo com valores elevados do limiar de aprendizado por reforço, indicando que a etapa Q-learning não foi tão relevante, nos casos das funções mais complexas houve crescimentos consideráveis dos percentuais de cobertura com a diminuição do valor da variável. Dessa forma, como valores maiores provaram ser em vezes suficientes porém valores menores foram extremamente necessários em alguns casos e não prejudiciais nos outros, o valor da variável de aprendizado por reforço seguiu como 0 para os experimentos finais. Todavia, vale ressaltar que se o tempo de execução fosse um parâmetro de maior peso na decisão, um valor maior para essa variável seria mais adequado.

Quadro 6 – Melhores valores encontrados para as funções

Função	Limite de gerações	Tamanho da população	Probabilidade de mutação	Limiar do aprendizado por reforço
primo	10	1	inconclusivo	0,35
triangle	10	1	inconclusivo	0,1
multiplos	100	10	90%	0,9
qtd_divisores	10	1	inconclusivo	0,1
repetidos	100	10	65%	0,35
faltante1	100	10	90%	0,35
uppCons	10	10	inconclusivo	0,65
freq_palavras	50	1	inconclusivo	0,1

O Quadro 6 indica os melhores valores encontrados nos experimentos das quatro variáveis de controle de execução para cada uma das oito funções selecionadas como SUT.

Finalmente, o resultado dos experimentos para encontrar os valores ideais para as variáveis de controle de execução do algoritmo MAAT implementado foram:

a) limite de gerações = 100

b) tamanho da população = 10

c) probabilidade de mutação = 100%

d) limiar do aprendizado por reforço = 0

4.2 COMPARAÇÃO MAAT X ALGORITMO GENÉTICO COMUM

Tendo descobertos os valores ideais para as variáveis de controle de execução, além de definir esses valores para os parâmetros do MAAT implementado, os mesmos também foram utilizados no algoritmo genético comum (GA), com exceção do limiar do aprendizado por reforço, visto que não existe essa etapa no algoritmo genético comum. Foram então realizados experimentos onde, com os mesmos valores de parâmetros, tanto o MAAT implementado quanto o algoritmo genético foram utilizados para gerar casos de teste automatizados para os códigos do conjunto de funções utilizadas nos experimentos anteriores.

Quadro 7 – Comparação dos algoritmos para a função primo

Algoritmo	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
MAAT	2,7	14	6	100%	100%
GA	58,2	100	57	88,64%	71,50%

Quadro 8 – Comparação dos algoritmos para a função triangle

Algoritmo	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
MAAT	1,9	2	9	100%	100%
GA	95,6	100	194	48.83%	9,1%

Quadro 9 – Comparação dos algoritmos para a função multiplos

Algoritmo	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
MAAT	40,8	100	69	95,15%	88,10%
GA	53,9	100	78	90,23%	77,20%

Quadro 10 – Comparação dos algoritmos para a função qtd_divisores

Algoritmo	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
MAAT	1,9	3	4	100%	100%
GA	27,1	100	21	98,47%	97,00%

Quadro 11 – Comparação dos algoritmos para a função repetidos

Algoritmo	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
MAAT	44,1	100	39	94,94%	88,00%
GA	86,8	100	55	67,00%	26,50%

Quadro 12 – Comparação dos algoritmos para a função faltante1

Algoritmo	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
MAAT	90,8	100	75	77,10%	18,60%
GA	92,6	100	58	54,00%	14,6%

Quadro 13 – Comparação dos algoritmos para a função uppCons

Algoritmo	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
MAAT	3,6	35	6	100%	100%
GA	31,3	100	23	97,30%	93,80%

Quadro 14 – Comparação dos algoritmos para a função freq_palavras

Algoritmo	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
MAAT	19,8	100	27	99,93%	99,80%
GA	100	100	67	41,27%	0%

Como evidenciado em todos os experimentos de comparação entre as performances do MAAT implementado e do algoritmo genético comum, o primeiro algoritmo apresenta superioridade incontestável nos percentuais de cobertura média e total do critério de conjuntos de nós em caminhos, o que já era esperado visto que o MAAT apenas realiza alterações em casos de teste redundantes dos cromossomos, consequentemente impossibilitando a redução do percentual de cobertura deles enquanto que o mesmo não ocorre no

outro algoritmo. Alguns casos que valem notar foram os que os percentuais de cobertura média e total do GA chegaram bem próximos ao do MAAT, mas ele alcançou 100% de cobertura enquanto o algoritmo comum não, cujos dados foram apresentados no Quadro 11 e no Quadro 13, e pelo outro lado, uma situação em que o percentual de cobertura total do MAAT beirou o 100% enquanto que o GA não alcançou a cobertura total em nenhuma das 1000 execuções, como mostrado no Quadro 14. Outro dado interessante a ser observado é que o MAAT alcançou 100% de cobertura total para metade das funções, já em contrapartida o GA não repetiu o feito para nenhuma das oito.

No quesito gerações, o MAAT teve a sua média sendo inferior para todas as funções, mas evidentemente o GA igualou o valor máximo em quatro casos, para as funções em que o MAAT não obteve o 100% de cobertura total. Para algumas funções, como a faltante1 exibida no Quadro 12, a média quase foi a mesma, porém para outras, como a triangle do Quadro 8, a diferença foi alta.

Em relação ao tempo de execução, por mais que a métrica não tenha sido tão relevante para o resto dos experimentos, chegando neste ponto havia a dúvida de se a evolução que as modificações do MAAT trazem para os cromossomos de cada geração compensaria o tempo extra gasto pelas mesmas. A única instância em que esse fato não se realizou foi para a função faltante1, onde o tempo médio por execução do MAAT superou o do GA, já em todos os outros experimentos de comparação a média do MAAT foi inferior, especialmente para a função triangle, onde o tempo por execução do GA foi quase 20 vezes maior.

Esses resultados podem ser levados como um indicativo de que, pelo menos em uma média de 1000 execuções, o algoritmo MAAT possui um bom desempenho em relação a um algoritmo genético comum, apresentando uma diferença considerável de número de gerações, tempo de execução e percentual de cobertura do critério de conjuntos de nós em caminhos.

Sobre o desempenho do algoritmo MAAT com a função faltante1 utilizada como SUT, esse foi o único caso em que a cobertura média esteve abaixo de 90% e percentual de cobertura total menor do que 80%. Analisando a lógica da função em si e sua CFG, que possui um conjunto de quatro requisitos de teste no critério de cobertura utilizado, é possível perceber que três desses quatro só são cobertos por casos de testes com valores de entrada muito específicos. Um dos requisitos só será satisfeito se o caso de teste for exatamente uma lista vazia, outro se o caso de teste for uma lista que contenha pelo menos um número e que contenha exatamente os números de 1 até o tamanho da lista e o último dos três se o caso de teste for uma lista que contenha pelo menos dois números, sendo um deles 1 e o resto algo diferente do que o conjunto de números entre 2 e o tamanho da lista. Como mostrado no Quadro 4, com um limite de 100 gerações, a cobertura média gira em torno de 75%, ou seja, provavelmente dois desses três casos de teste mais específicos são encontrados, enquanto que um deles fica faltando. O experimento foi refeito com

os valores ideais das variáveis de controle de execução porém aumentando o limite de gerações para 500 e depois 1000, como exibido no Quadro 15 e conclui-se que realmente o limite de 100 gerações estava prejudicando o desempenho do algoritmo para a geração de um conjunto de casos de teste que atingisse a cobertura total.

Quadro 15 – Aumentando o limite de gerações para a função faltante1

Limite de gerações	Média de gerações	Máximo de gerações	Tempo médio (milissegundos)	Cobertura média	Percentual de cobertura total
500	327,3	500	266	89,15%	60,70%
1000	460,4	1000	378	96,10%	85,80%

5 CONCLUSÃO

No contexto do ensino introdutório de programação, educadores podem ter à sua disposição ferramentas para automatizar o processo de teste de software dos programas desenvolvidos pelos alunos. Para tal, essas ferramentas precisam ter casos de teste que possam avaliar o funcionamento do código solução dos alunos perante o problema abordado, ou seja, é necessário que seja gerado um conjunto de dados de entrada que consiga explorar as diferentes possibilidades de execução cabíveis ao código para posteriormente comparar os resultados obtidos com os esperados. Uma categoria de algoritmos utilizados para a geração automatizada de casos de teste são os algoritmos genéticos e variações modificadas deles. Com o objetivo de aperfeiçoar o projeto de testes de ferramentas de ensino, aprimorando o aprendizado de alunos de introdução à programação, é possível que a utilização de um algoritmo genético modificado maximize a performance da geração de casos de teste em comparação a um algoritmo genético comum.

Dessa forma, foi realizada uma pesquisa almejando encontrar artigos acadêmicos que abordassem soluções para a geração de casos de teste utilizando algoritmos genéticos. Dentre diversas propostas interessantes, a que se destacou foi a do artigo "Automation of software test data generation using genetic algorithm and reinforcement learning" (ESNA-ASHARI; DAMIA, 2021), onde foi detalhado o desenvolvimento do Memetic Algorithm for Automatic Test case generation (MAAT), um algoritmo genético gerador de casos de teste que implementa uma etapa de aprendizado por reforço e uma de busca adicional, além de só realizar alterações em casos de teste redundantes. Como o código fonte do algoritmo não foi disponibilizado, houve a necessidade de implementá-lo seguindo o pseudocódigo e suas explicações relatadas no artigo de origem. O algoritmo utilizado neste trabalho foi então desenvolvido em Python de forma a gerar casos de teste para funções com parâmetros de entrada de tipo lista de inteiros e string também, diferente do MAAT original que só aceitava funções recebendo inteiros. A partir do MAAT implementado, foi derivado um algoritmo genético comum para comparação, sendo ele uma versão do algoritmo porém com a exclusão das modificações desenvolvidas para o MAAT.

No intuito de aumentar o desempenho do MAAT implementado, foram realizados experimentos para encontrar os valores ideais das suas variáveis de controle de execução. Para esses experimentos, foram utilizados como SUT códigos de introdução à programação com diferentes tipos de parâmetros de entrada aceitos pelo MAAT implementado e níveis de complexidade suficiente para que os resultados dos experimentos fossem conclusivos. Tendo encontrado os valores ideais, os mesmos foram definidos para as variáveis de controle de execução do MAAT implementado e do algoritmo genético comum derivado dele. Dessa forma, foi possível realizar experimentos comparando o desempenho dos dois algoritmos nas métricas de número de iterações realizadas, tempo de execução e

percentual de cobertura alcançado.

Ao analisar os dados resultantes desses experimentos de comparação de performance dos dois algoritmos, foi possível perceber que o MAAT implementado atinge percentuais de cobertura média e total com superioridade em relação ao algoritmo genético comum ao mesmo tempo que necessita de menos iterações realizadas para atingir a cobertura total e gasta menos tempo para chegar ao resultado da execução. Assim sendo, conclui-se que no contexto da automatização da geração de casos de teste para códigos de introdução à programação, o algoritmo MAAT possui melhor eficiência do que um algoritmo genético comum.

Para trabalhos futuros, seria interessante generalizar a lógica do algoritmo para poder gerar casos de teste para funções com quaisquer tipos de parâmetros de entrada e considerar outras possibilidades de tipos de cobertura por grafos a serem utilizadas para a etapa de avaliação de aptidão dos cromossomos. Além disso poderiam ser consideradas e avaliadas outras opções de alteração dos valores dos casos de teste dos cromossomos para as etapas de mutação e aprendizado por reforço, dependendo do tipo dos parâmetros de entrada do SUT. Também para a etapa de aprendizado por reforço, poderia ser analisado o impacto de cada um dos oito tipos de alteração na solução obtida. Ademais, poderiam ser avaliadas as escolhas de modificações do algoritmo implementado em relação ao original, desenvolvendo um algoritmo mais fiel ao detalhado no artigo. Para os experimentos, poderia ser levada em conta a métrica de número de chamadas de avaliação de função que o algoritmo necessita. Finalmente, com a integração do algoritmo com alguma ferramenta geradora de CFGs e que encontre os caminhos do grafo o MAAT poderia ser utilizado como função geradora de casos de teste para as ferramentas de auxílio ao ensino da programação.

REFERÊNCIAS

ALANDER, J. T.; MANTERE, T.; TURUNEN, P. Genetic algorithm based software testing. **Proceedings of the International Conference**, 1997.

ALBUQUERQUE, A. M. d.; BOECHAT, F. A.; COUTINHO, L. S. **Modelagem de testes de software: uma análise dos resultados de testes em exercícios de programação**. Dissertação (Mestrado) — Universidade Federal do Rio de Janeiro, 2023.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. [S.l.]: Cambridge University Press, 2016.

BERNDT, D. et al. Breeding software test cases with genetic algorithms. **Hawaii International Conference on System Sciences**, 2003.

BERNDT, D.; WATKINS, A. Genetic algorithm based software testing. **Hawaii International Conference on System Sciences**, 2005.

BOUCHACHIA, A. An immune genetic algorithm for software test data generation. **Seventh International Conference on Hybrid Intelligent Systems**, 2007.

DAMIA, A.; PARVIZIMOSAED, M. Software testing using an adaptive genetic algorithm. **Journal of Artificial Intelligence and Data Mining (JAIDM)**, v. 9, 2021.

ELECTRICAL, I. of; ENGINEERS, E. **1012-2016 - IEEE Standard for System, Software, and Hardware Verification and Validation**. [S.l.]: IEEE, 2017.

ESNAASHARI, M.; DAMIA, A. Automation of software test data generation using genetic algorithm and reinforcement learning. **Expert Systems With Applications**, v. 183, 2021.

GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization, and Machine Learning**. [S.l.]: Addison-Wesley, 1989.

SHARMA, A.; PATANI, R.; AGGARWAL, A. Software testing using genetic algorithms. **International Journal of Computer Science Engineering Survey (IJCSES)**, v. 7, 2016.

SRIVASTAVA, P. R. Application of genetic algorithm in software testing. **International Journal of Software Engineering and Its Applications**, v. 3, 2009.

APÊNDICE A – EXPERIMENTOS ALTERANDO O LIMITE DE GERAÇÕES.

função	limite de gerações	máximo de gerações	média de gerações	tempo médio de execução (milissegundos)	cobertura média	percentual de cobertura total
primo	1	1	1	4	73,46%	0%
	10	10	2,7	6	99,80%	98,70%
	50	31	2,8	6	100,00%	100,00%
	100	16	2,7	6	100,00%	100,00%
triangle	1	1	1	7	70,37%	0%
	10	2	1,9	9	100,00%	100,00%
	50	3	1,9	9	100,00%	100,00%
	100	3	1,9	9	100,00%	100,00%
multiplos	1	1	1	4	61,23%	0,00%
	10	10	8,9	18	68,03%	18,80%
	50	50	31,5	62	85,13%	62,80%
	100	100	43,1	80	94,35%	86,10%
qtd_divisores	1	1	1	3	71%	0%
	10	4	1,9	4	100%	100%
	50	3	1,9	5	100%	100%
	100	3	1,9	4	100%	100%
repetidos	1	1	1	2	51,53%	0%
	10	10	9,6	11	61,10%	10%
	50	50	33	33	85,53%	66%
	100	100	42,4	42	95,95%	90,50%
faltante1	1	1	1	2	51,90%	0%
	10	10	9,97	10	63,50%	0,60%
	50	50	48,3	47	74,05%	7,40%
	100	100	92,3	86	76,53%	16,60%
uppCons	1	1	1	3	76,03%	0%
	10	10	3,1	6	96,40%	90,80%
	50	43	3,7	7	100%	100%
	100	31	3,6	7	100%	100%
freq_palavras	1	1	1	3	48,53%	0%
	10	10	8,5	14	75,60%	32,90%
	50	50	17,9	28	98%	93,50%
	100	100	18,3	28	99,93%	99,80%

Dados dos experimentos realizados para analisar o comportamento da performance do algoritmo MAAT implementado tendo o valor do seu limite de gerações alterado.

APÊNDICE B – EXPERIMENTOS ALTERANDO O TAMANHO DA POPULAÇÃO.

função	tamanho da população	média de gerações	máximo de gerações	tempo médio de execução (milissegundos)	cobertura média	percentual de cobertura total
primo	1	3,3	27	1	100%	100%
	10	2,7	14	6	100%	100%
	50	2,6	16	25	100%	100%
	100	2,8	16	52	100%	100%
triangle	1	1,9	2	0,7	100%	100%
	10	1,9	2	8	100%	100%
	50	1,9	2	43	100%	100%
	100	1,9	3	85	100%	100%
multiplos	1	70,5	100	8	76,95%	51,80%
	10	42,7	100	72	94,18%	85,70%
	50	40,5	100	354	95,15%	87%
	100	42,3	100	722	94,60%	86,90%
qtd_divisores	1	1,9	3	0,7	100%	100%
	10	1,9	3	4	100%	100%
	50	1,9	3	18	100%	100%
	100	1,9	4	37	100%	100%
repetidos	1	69,8	100	8	76,15%	53,30%
	10	42,7	100	38	95,15%	88,70%
	50	39,2	100	164	97,03%	92,50%
	100	40,7	100	343	95,85%	90,30%
faltante1	1	94,5	100	10	71,18%	11%
	10	91,3	100	76	77,13%	18,30%
	50	91,1	100	356	76,93%	18,10%
	100	90,38	100	714	77,80%	18,20%
uppCons	1	14	100	7	98,40%	95,20%
	10	3,8	34	6	100%	100%
	50	3,5	31	22	100%	100%
	100	3,2	25	41	100%	100%
freq_palavras	1	25,5	100	15	98,70%	96%
	10	18,5	100	25	99,93%	99,80%
	50	19,5	100	91	99,80%	99,30%
	100	19,1	100	171	99,90%	99,70%

Dados dos experimentos realizados para analisar o comportamento da performance do algoritmo MAAT implementado tendo o valor do seu tamanho de população alterado.

APÊNDICE C – EXPERIMENTOS ALTERANDO A PROBABILIDADE DE MUTAÇÃO.

função	probabilidade de mutação	média de gerações	máximo de gerações	tempo médio de execução (milissegundos)	cobertura média	percentual de cobertura total
primo	10%	3,1	16	6	100%	100%
	35%	3	22	6	100%	100%
	65%	2,8	25	6	100%	100%
	90%	2,7	18	6	100%	100%
triangle	10%	1,9	3	9	100%	100%
	35%	1,9	2	8	100%	100%
	65%	1,9	3	8	100%	100%
	90%	1,9	2	8	100%	100%
multiplos	10%	54,6	100	90	91,63%	72,50%
	35%	53,2	100	88	90,60%	72,90%
	65%	48,3	100	80	92,88%	81,50%
	90%	46,5	100	77	92,88%	81,70%
qtd_divisores	10%	1,9	5	4	100%	100%
	35%	1,9	4	4	100%	100%
	65%	1,9	3	4	100%	100%
	90%	1,9	3	4	100%	100%
repetidos	10%	58,7	100	47	91,13%	73,30%
	35%	46,7	100	39	94,85%	86,50%
	65%	43,4	100	37	96,25%	90,10%
	90%	44	100	39	95,08%	89,30%
faltante1	10%	97,7	100	74	74,28%	5%
	35%	95,5	100	75	75,53%	9%
	65%	93,84	100	76	76,40%	12,20%
	90%	90,7	100	79	77,43%	18,40%
uppCons	10%	4,1	39	6	100%	100%
	35%	4	38	6	100%	100%
	65%	3,7	37	6	100%	100%
	90%	3,7	34	6	100%	100%
freq_palavras	10%	20	100	26	99,93%	99,80%
	35%	19,7	100	26	99,87%	99,50%
	65%	19,8	100	26	99,87%	99,60%
	90%	19,2	100	26	99,90%	99,70%

Dados dos experimentos realizados para analisar o comportamento da performance do algoritmo MAAT implementado tendo o valor da sua probabilidade de mutação alterado.

APÊNDICE D – EXPERIMENTOS ALTERANDO O LIMIAR DO APRENDIZADO POR REFORÇO.

função	1,00 - limiar do aprendizado por reforço	média de gerações	máximo de gerações	tempo médio de execução (milissegundos)	cobertura média	percentua de cobertura total
primo	0,1	58,5	100	71	94,56%	78,40%
	0,35	19	100	25	99,92%	99,80%
	0,65	2,7	19	5	100%	100%
	0,9	2,7	19	5	100%	100%
triangle	0,1	99,2	100	213	62,83%	1,70%
	0,35	22,4	100	51	99,13%	98,60%
	0,65	22,8	100	51	99%	98,50%
	0,9	1,9	3	9	100%	100%
multiplos	0,1	42	100	67	96,73%	89,20%
	0,35	45,6	100	74	93,03%	83%
	0,65	47,4	100	79	91,53%	80,30%
	0,9	42,3	100	72	94,63%	85,70%
qtd_divisores	0,1	54,5	100	73	93,30%	82,20%
	0,35	16,4	100	24	99,67%	99,50%
	0,65	17	100	25	99,80%	99,70%
	0,9	1,9	4	4	100%	100%
repetidos	0,1	92,6	100	73	72,93%	14,20%
	0,35	94,7	100	76	56,28%	10,60%
	0,65	44,5	100	39	95,40%	89,70%
	0,9	43,5	100	38	94,73%	88,40%
faltante1	0,1	93,4	100	72	75%	12,90%
	0,35	92,5	100	72	60,58%	13,90%
	0,65	90,4	100	75	77,18%	18,70%
	0,9	91,9	100	77	77,15%	15,50%
uppCons	0,1	39,7	100	35	94,07%	82,30%
	0,35	4,2	38	6	100%	100%
	0,65	3,8	49	5	100%	100%
	0,9	3,8	29	6	100%	100%
freq_palavras	0,1	99,9	100	76	65,37%	0,10%
	0,35	38,9	100	39	96,50%	92,80%
	0,65	38	100	38	96,64%	92,70%
	0,9	18,8	100	25	99,90%	99,70%

Dados dos experimentos realizados para analisar o comportamento da performance do algoritmo MAAT implementado tendo o valor do seu limiar de aprendizado por reforço alterado.