Uma avaliação de técnicas e critérios de teste de software para a linguagem de programação Python

Renata O. Brito, Stevão A. Andrade, Márcio E. Delamaro

[1]Universidade de São Paulo (USP)
Instituto de Ciências Matemáticas e de Computação (ICMC)
Avenida Trabalhador São-carlense, 400, CEP: 13566-590 - Centro – São Carlos – SP – Brasil

{renata.oliveira.brito@usp.br, {stevao,delamaro}@icmc.usp.br

Abstract. *The Python programming language has gained ground in the software industry and has become one of the most popular programming languages, mostly due to its simplicity and flexibility, which facilitates learning, in addition to popularizing the use of new computing technologies, such as machine learning. Considering this scenario, this work explores this problem by categorizing and evaluating software testing techniques and criteria applied to the Python programming language. Among the activities developed, it is worth highlighting an assessment regarding the ability to apply the testing techniques context of Python, categorization of tools and technologies that support the application of such approaches, as well as the development of a practical study to measure the feasibility of applying them in the context of open-source projects.*

Resumo. *A linguagem de programação Python tem ganhado espaço na indústria de software e ter se tornado uma das linguagens de programação mais populares, devido sua simplicidade e flexibilidade, que facilita o aprendizado, além de ter popularizado a utilização de novas tecnologias de computação, como, por exemplo, aprendizado de máquina. Considerando esse cenário, esse projeto explora tal problemática por meio da categorização e avaliação de técnicas e critérios de teste de software aplicados a linguagem de programação Python. Dentre as atividades desenvolvidas, destaca-se uma avaliação a respeito da capacidade de aplicação das técnicas de teste contexto da linguagem de programação Python, categorização de ferramentas/tecnologias que dão suporte à aplicação de tais abordagens, bem como a realização de um estudo prático de viabilidade de aplicação das mesmas no contexto de projetos open-source.*

1. Introdução

A linguagem de programação *Python* ganhou espaço na indústria de software e tem se tornado uma das linguagens de programação mais populares devido sua simplicidade e flexibilidade, que facilitam o aprendizado, além de ter popularizado a utilização de novas tecnologias de computação, como, por exemplo, aprendizado de máquina [Srinath 2017].

Entretanto, conforme preconizado pela engenharia de software, a popularização de novas linguagens de programação e novas tecnologias também deve ser acompanhada pela popularização de ferramentas e técnicas que visem garantir a qualidade no processo de construção de software. Segundo Pressman [Pressman 2010], a engenharia de software

é definida como sendo uma área de estudo que tem como finalidade aplicar métodos científicos em benefício da produção de software com objetivo de produzir produtos com baixo custo e alta qualidade. Considerando esse cenário, esse trabalho visou explorar essa problemática por meio da categorização e avaliação de técnicas e critérios de teste de software aplicados à linguagem de programação *Python*.

Apesar do rápido desenvolvimento de novas linguagens de programação, impulsionadas por novas funcionalidades e formas mais intuitivas e produtivas para solucionar problemas, nem todo o domínio de computação é capaz de absolver essas novas tendências e disponibilizar o conjunto de ferramentas adequadas para dar suporte à aplicação de conceitos fundamentais para o sucesso de um projeto [Gilsing et al. 2011]. Além das dificuldades já enfrentadas em tecnologias consolidadas, um dos grandes desafios a serem enfrentados por novas tecnologias é a disponibilização de um conjunto ferramental que atenda aos requisitos técnicos/científicos para dar suporte à utilização das mesmas. Nesse sentido, esse trabalho busca contribuir ao avaliar o material ferramental, que dê suporte à prática de teste de software para *Python*, além de fornecer material para dar suporte à avaliação experimental de novas abordagens de teste para o domínio da linguagem.

Em síntese, foram catalogadas ferramentas de teste de software existentes para a linguagem de programação *Python*, que implementam critérios de teste de software estrutural e teste de mutação. Após o levantamento foi conduzida uma avaliação experimental[1] para aferir a eficácia de tais ferramentas no que diz respeito à aderência das mesmas aos conceitos teóricos desenvolvidos na área de teste de software.

Para realizar o processo de avaliação das ferramentas de teste foi utilizado um popular *beanchmark* de programas. Como parte dos resultados deste trabalho, além da avaliação das ferramentas, o conjunto de testes existente no *beanchmark* foi aprimorado para atender os critérios de teste explorados nesse trabalho.

As demais seções deste documento descrevem os procedimentos adotados durante a condução do trabalho. A Seção 2 descreve os principais objetivos deste trabalho, bem como a metodologia adotada durante o desenvolvimento do mesmo. A Seção 3 apresenta um resumo dos resultados sobre o estudo da linguagem de programação *Python*, além da revisão bibliográfica sobre os tópicos de teste de software atrelados ao tema da pesquisa. A Seção 4 apresenta informações sobre o *benchmark* utilizado para a condução da avaliação experimental, a Seção 5 apresenta as ferramentas de teste de software selecionadas para o desenvolvimento desse projeto, a Seção 6 apresenta os resultados das avaliações experimentais conduzidas durante todo o projeto e, finalmente, a Seção 7 apresenta as conclusões e os principais aprendizados obtidos durante o desenvolvimento deste trabalho.

2. Metodologia

Esse trabalho teve como principal objetivo investigar a disponibilidade de ferramentas de teste de software disponíveis para o domínio da linguagem de programação *Python*. Além disso, avaliar a eficácia de tais ferramentas no que diz respeito à aderência das mesmas aos conceitos teóricos desenvolvidos na área de teste de software.

[1]Todos os artefatos desenvolvidos encontram-se disponíveis publicamente no repositório de código aberto `https://github.com/RenataBrito/QuixBugs_report`

Em síntese, o trabalho catalogou ferramentas capazes de aplicar os conceitos de teste de software, especificamente referentes aos critérios de teste de software estrutural e teste de mutação. A partir do levantamento das ferramentas foi escolhido um *benchmark* de programas, disponíveis em repositórios *open-source*, com o intuito de verificar se as ferramentas atendem à demanda dos especialistas que fazem uso de tal linguagem de programação.

Em virtude do material produzido, foram adicionados objetivos específicos que observaram a adequação do conjunto de casos de teste presente no *benchmark* utilizado, possibilitando a adequação do mesmo aos critérios de teste investigados, e a avaliação das ferramentas de teste de mutação quanto a sua eficiência em termos de tempo de execução, uma vez que esse é um fator conhecidamente como limitante na aplicação da técnica.

Considerando o objetivo do trabalho, as seguintes atividades foram desenvolvidas:

1. **Revisão bibliográfica sobre conceitos da linguagem de programação *Python***: o foco dessa atividade é entender o funcionamento da linguagem de programação e verificar os principais mecanismos da mesma que se diferem em relação às demais linguagens de programação e que podem portanto impactar no uso de ferramentas de teste.
2. **Revisão bibliográfica sobre conceitos de Teste de Software**: especificamente sobre as técnicas de teste estrutural e teste de mutação, de forma a entender como conceitos podem ser aplicados ao contexto de ferramentas de teste de software disponíveis para a linguagem de programação *Python*.
3. **Catalogar ferramentas de teste de software em *Python***: essa etapa correspondeu ao levantamento de ferramentas/tecnologias que dão suporte a aplicação de teste de software utilizando a linguagem de programação *Python*.
4. **Catalogar projetos em *Python* disponíveis em repositórios *open-source***: definição de projetos disponíveis em repositórios *open-source*, com intuito de atestar a aplicabilidade das ferramentas levantadas na etapa anterior.
5. **Avaliação experimental**: considerando os projetos levantados na etapa anterior, as ferramentas foram avaliadas e os resultados referentes às dificuldades e limitações das ferramentas foram apresentados.
6. **Aprimoramento do *benchmark* de programas com conjuntos de casos de teste adequados aos critérios de teste:** a consolidação dos resultados dos objetivos anteriores permitiu o melhoramento do conjunto de casos de teste disponíveis no *benchmark* utilizado para avaliar as ferramentas, de tal forma que o novo *benchmark* é adequado aos critérios de teste de software investigados neste trabalho.
7. **Avaliação do custo computacional de execução para ferramentas de teste de mutação**: uma vez de posse do *benchmark* adequado aos critérios de teste investigados, nós avaliamos como as ferramentas de teste de mutação se comportam quanto ao seu custo computacional de execução.

De forma geral, o desenvolvimento das etapas seguiu um processo sequencial, descrito na Figura 1, no qual o desenvolvimento da atividade seguinte exigia, obrigatoriamente, a conclusão da atividade anterior.

O processo de aprimoramento do *benchmark* foi realizado por meio da avaliação individual de cada programa contido no *benchmark*, seguindo um processo de teste inte-

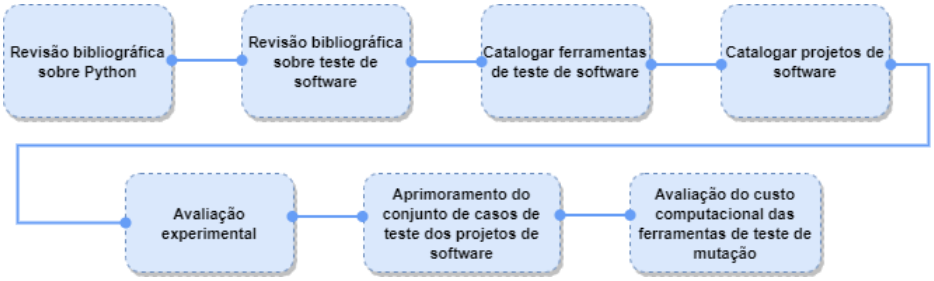

Figura 1. *Workflow* das atividades desenvolvidas durante a pesquisa

rativo, utilizando como critério de conclusão a adequação do conjunto de casos de testes de acordo com os critérios de teste estrutural e de mutação.

3. Revisão bibliográfica

3.1. A linguagem de programação *Python*

As origens da linguagem de programação *Python* devem-se a um projeto iniciado em 1989 por Guido van Rossum[2], um matemático e cientista da computação holandês, formado pela Universidade de Amsterdã, devido a deficiências existentes em outras linguagens de programação [Van Rossum et al. 2007].

De acordo com o gráfico apresentado na Figura 2, a linguagem de programação *Python*, 32 anos depois de sua criação, tem se tornado cada vez mais popular em comparação às demais linguagens de programação. O número de buscas a respeito dessa linguagem de programação triplicou desde 2010 se tornando umas das maiores pesquisas do *Google* nesse período [The economist 2018], enquanto as demais linguagens permaneceram estáveis ou diminuíram.

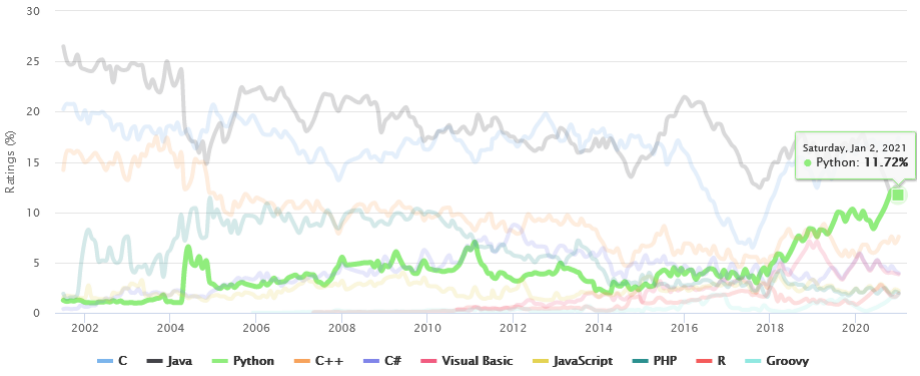

Figura 2. Ranking com as linguagens de programação mais populares de 2021 segundo a TIOBE [Tiobe the software quality company 2021].

Dentre os motivos para a popularização da linguagem de programação *Python*, destaca-se a sua versatilidade e a gama de recursos que a linguagem de programação possui, como sistemas de tipo dinâmico, que permitem que os desenvolvedores escrevam muito menos linhas de código para tarefas que exigem mais linhas de código em outras linguagens. Isso torna a linguagem de programação muito fácil de aprender, mesmo

[2]https://pt.wikipedia.org/wiki/Guido_van_Rossum

para iniciantes. *Python* também é muito famosa por sua sintaxe de programação simples, legibilidade de código e comandos semelhantes ao inglês que tornam a codificação em *Python* muito mais fácil e eficiente. Além disso, a linguagem possui um rico conjunto de bibliotecas e *frameworks* para diversos fins, como *big data*, *web development*, *machine learning*, *cloud computing*, *scientific computing* e etc [Millman and Aivazis 2011].

3.2. Teste de Software

Teste de software é uma atividade dinâmica, que tem como principal objetivo executar um dado programa utilizando um conjunto de entradas específicas e identificar se seu comportamento está de acordo com sua especificação ou com o objetivo para o qual foi construído. Caso a execução apresente resultados não esperados, indica-se que foi revelada uma falha no programa e é necessária uma nova etapa de correção, portanto, define-se a atividade de teste de software como um processo, ou uma série de processos, que tem como finalidade executar um dado programa com a intenção de revelar possíveis defeitos existentes [Delamaro et al. 2016a].

De acordo com estudos recentes, a atividade de teste de software está entre aquelas que consome maior parte no ciclo de desenvolvimento de software, tornando-se assim em uma das etapas mais onerosas desse processo [Hynninen et al. 2018]. Portanto, a adoção de práticas para a automatização da atividade de teste de software contribui significativamente para a redução tanto nos custos como no tempo de desenvolvimento de um projeto de software [Sandler et al. 2012].

Lidar com o tamanho do domínio de entradas possíveis para se testar um programa é um grande desafio, uma vez que mesmo para programas simples, o número de entradas existentes pode ser considerado muito grande e em alguns casos, até infinito [Delamaro et al. 2016a]. Portanto, é necessário definir mecanismos capazes de particionar o domínio de entradas para testar um programa, tais abordagens são definidas como técnicas de teste de software e as duas investigadas nesse trabalho são descritas nas subseções abaixo.

3.2.1. Teste Estrutural

Na técnica de teste estrutural os requisitos de teste são extraídos exclusivamente a partir de informações e características do código-fonte do programa em teste, assim o testador pode analisar o comportamento de rotinas especificas do programa em teste, permitindo que haja uma verificação mais profunda do seu comportamento [Barbosa et al. 2016].

Os casos de teste são criados a partir dos detalhes da estrutura interna do código-fonte do programa. A partir disso são definidos um conjunto de elementos de software que devem ser executados para que se atinja a cobertura mínima para um determinado critério.

Os critérios desta técnica utilizam uma representação do programa conhecida como grafo de fluxo de controle (GFC). O GFC serve para que possamos fazer uma abstração da estrutura do programa e os critérios de teste dessa técnica medem o grau de cobertura dos casos de teste em relação ao grafo, como, por exemplo, os critérios “Todos-nós”e “Todas-arestas”[Barbosa et al. 2016].

3.3. Teste Baseado em Defeitos - Teste de Mutação

O teste de mutação é critério baseado defeitos, ou seja, utiliza conhecimento sobre defeitos típicos que podem ocorrer ao escrevermos programa. Com base nessa atividade nos ajuda a criar bons conjuntos de testes capazes de detectar os problemas modelados a partir dos defeitos modelados de forma artificial [Delamaro et al. 2016b]. A teoria por trás do teste de mutação atesta que casos de teste capazes de identificar falhas ocasionadas a partir de defeitos simples são tão sensíveis que, implicitamente, também são capazes de revelar outros tipos de falhas, devido a uma característica conhecida como efeito de acoplamento, que foi evidenciada por meio de experimentos científicos [Offutt 1992].

Os programas mutantes são criados a partir dos operadores de mutação, tais programas mutantes são gerados com objetivo de modelar defeitos comuns que podem ser erroneamente inseridos durante o processo de desenvolvimento de software, por conta de um engano cometido pelo desenvolvedor durante o processo de codificação. Os operadores de mutação são as regras que definem as alterações a serem aplicadas a um determinado programa, criando assim programas similares ao original (mutantes). Os operadores de mutação podem variar de acordo com a linguagem de programação do programa, gerados a partir de pequenas variações em estruturas de código como comandos, variáveis, constantes e operadores relacionais [Delamaro et al. 2016b].

Ao executar um dado caso de teste, se a saída do programa original for diferente da saída produzida por um dado mutante **A**, denomina-se que o mutante **A** foi *morto*, ou seja, o caso de teste projetado foi capaz de revelar a diferença entre o programa mutante e o seu programa original. Porém, se as execuções forem iguais, é necessário observar o comportamento do programa mutante e determinar se é possível criar um caso de teste capaz de demonstrar a diferença existente entre o programa original e o mutante. Caso não seja possível criar um caso de teste capaz de identificar essa diferença então é dito que o programa mutante é *equivalente* ao programa original.

A dinâmica do teste de mutação estende-se em criar novos casos de teste com base no conjunto de mutantes existentes, de forma a verificar o programa em teste, até que todos os mutantes sejam *mortos* e os demais sejam identificados como *equivalentes*.

4. Catálogo de projetos

Foi utilizado um conjunto de 40 programas (descritos na Tabela 1) escritos em *Python*, cada programa contendo apenas um único defeito em uma única linha. O conjunto de programas que compõe esse *benchmark* é chamado de *QuixBugs* [Lin et al. 2017] e é baseado em problemas do *Quixey Challenge*, uma competição na qual programadores tinham como desafio receber um programa curto contendo um defeito e ter 1 minuto para identificar e corrigir tal problema.

O repositório que contém os dados desse projeto possui um documento que descreve a funcionalidade de cada um dos 40 programas utilizados neste trabalho[3].

A vantagem de utilizar tal *benchmark* é o fato de que de antemão temos acesso a um conjunto representativo de programas que possui um conjunto inicial de casos de teste que é capaz de revelar os defeitos existentes nos programas. Além disso, o *benchmark*

[3] `https://github.com/RenataBrito/QuixBugs_report/blob/master/Results/descricaoDosProgramas.pdf`

Tabela 1. Programas contidos no *benchmark*

bitcount	breadth first search	bucketsort	depth first search
detect cycle	find first in sorted	find in sorted	flatten
gcd	get factors	hanoi	is valid parenthesization
kheapsort	knapsack	kth	lcs length
levenshtein	lis	longest common subsequence	max sublist sum
mergsort	minimum spanning tree	next palindrome	net permutation
pascal	possible change	powerset	quicksort
reverse linked list	rpn eval	shortest path length	shortest path lengths
shortest paths	shunting yard	sieve	sqrt
subsequences	to base	topological ordering	wrap

conta tanto com as versões dos programas com defeitos quanto a sua versão corrigida, possibilitando que análises experimentais mais profundas possam ser realizadas.

Os defeitos inseridos dentro do *benchmark* são classificados em 8 tipos, conforme descrito na tabela abaixo:

Tabela 2. Classes de defeitos

Classe de defeitos	**Número**
Operador de atribuição incorreto	1
Variável incorreta	5
Operador de comparação incorreto	5
Condição ausente	2
Incremento ausente / adicionado	4
Inversão de variáveis	6
Divisão incorreta de vetores	2
Variáveis anexadas	2
Constante de estrutura de dados incorreta	2
Método incorreto chamado	1
Desreferenciação de campo incorreta	1
Expressão aritmética ausente	1
Chamada de função ausente	4
Linha ausente	4

5. Ferramentas de teste para *Python*

Essa etapa corresponde ao levantamento de ferramentas/tecnologias que podem dar suporte à aplicação de práticas de teste de software utilizando a linguagem de programação *Python*.

5.1. Teste estrutural

Uma das principais das tecnologias mantidas para dar suporte à aplicação de teste estrutural é a biblioteca *pytest*, descrita com maiores detalhes abaixo:

- ***pytest*** - podendo ser utilizado em *Python* 2 e *Python* 3, *pytest* [Krekel 2021] é um *framework* baseado em *Python*, uma estrutura de teste de software, o que significa que o *pytest* é uma ferramenta de linha de comando que localiza automaticamente

os testes que foram escritos, executa os testes e relata os resultados, usada para todos os tipos e níveis de teste de software. É possível utilizar *plugins* podendo obter uma maior eficácia na cobertura.
Dentre as principais características existentes que motivam a utilização do *pytest*, é possível destacar que a ferramenta permite a criação de suítes de teste compactas, disponibiliza informações precisas sobre falhas de execução, possui baixa dependência de convenções de código, *fixtures* são simples e fáceis de usar, é possível parametrizar diferentes testes, além de possuir muitos *plugins* disponíveis.

5.2. Teste de mutação

Para teste de mutação foram identificadas três ferramentas que possuem desenvolvimento ativo, são elas *cosmic-ray*, *mutmut* e *mutpy*. As ferramentas são descritas com maiores detalhes abaixo:

- ***cosmic-ray*** - desenvolvido por Austin Abingham e Robert Smallshire [Abingham and Smallshire 2021], a ferramenta de teste de mutação *cosmic-ray* para *Python* 3, faz pequenas alterações no código-fonte, executando seu conjunto de testes para cada um, e está sendo desenvolvida ativamente.
  O conjunto de mutantes gerados pela ferramenta é baseado em 14 operadores de mutação [Sixty North AS].
  A ferramenta *cosmic-ray* usa uma noção de sessões para atingir um conjunto completo de testes de mutação. Estas sessões armazenam dados sobre o andamento de uma execução. Os dados da sessão são persistidos em um banco de dados utilizando a biblioteca *sqlite*, por meio do qual é possível posteriormente consultar os relatórios e resultados referentes à sessão de teste.
- ***mutmut*** - desenvolvido por Anders Hovmöller [Hovmöller 2021] a partir da dificuldade que encontrou em utilizar as outras ferramentas já existente. Anders Hovmöller propôs uma ferramenta de teste de mutação *mutmut* que está disponível tanto para *Python* 2, quanto para *Python* 3. A ferramenta tem como principal foco a sua praticidade e facilidade no uso, apontando como principal vantagem a capacidade de gerar mutações direto no arquivo de origem, de forma a evitar a criação de arquivos de configuração, ou arquivos de cópia desnecessários no diretório do projeto. A ferramenta conta com 10 operadores de mutação.
  Dentre os principais objetivos focados na implementação da ferramenta, destaca-se a possibilidade de lidar com mutantes e realizar operações de forma individualizada e a execução por meio de linha de comando.
- ***mutpy*** - desenvolvida por Konrad Halas como parte de uma dissertação de mestrado [Hałas 2014], a ferramenta de teste de mutação *Mutpy* suporta código-fonte *Python* 3.3+.
  A ferramenta suporta as bibliotecas *pytest* e *unittest* para execução dos casos de teste. Possui 20 operadores de mutação padrões e mais uma lista de 7 operadores de mutação experimentais [Konrad Halas].

6. Avaliação

O objetivo inicial dessa atividade visava verificar o estado atual do conjunto de casos de teste previamente existente no *benchmark* e posteriormente incrementar o *benchmark*

com novos casos de teste e avaliar o comportamento das ferramentas de teste com relação às suas funcionalidades e limitações.

Para cada uma das ferramentas investigadas o *benchmark* selecionado foi executado com um conjunto de casos de teste projetados especificamente para cada programa. Para facilitar a análise dos resultados também foi incluído um *script* de teste, que funciona como um *test runner*. Esse *script* é responsável por fazer a chamada da ferramenta, executar todos os casos de testes existentes e controlar a execução e a coleta dos resultados.

O repositório do projeto apresenta o conjunto de casos de teste adequado aos critérios de teste investigados neste trabalho e nas subseções seguintes nós apresentamos algumas percepções observadas após o uso de cada uma das ferramentas analisadas nesse projeto.

6.1. *pytest*

Com base nos resultados obtidos no relatório de cobertura [4] foi possível concluir que a maioria dos programas apresentaram um bom conjunto inicial de caso de teste, chegando à cobertura de 100% para o critério de teste estrutural todas-arestas, porém alguns programas tiveram exceções para chegar a esse resultado esperado e foi necessária a inclusão de novos casos de teste para atingir o percentual de cobertura ideal. A Tabela 3 apresenta os resultados para todos os programas após a inclusão dos casos de teste para adequação do conjunto de casos de teste ao critério todas-arestas.

Program	Statments	Miss	Branch	Branch Partial	Coverage
bitcount	6	0	2	0	100%
breadth_first_search	13	0	6	0	100%
bucketsort	8	0	4	0	100%
depth_first_search	10	0	6	0	100%
detect_cycle	8	0	4	0	100%
find_first_in_sorted	11	0	6	0	100%
find_in_sorted	11	0	6	0	100%
flatten	6	0	6	0	100%
gcd	4	0	2	0	100%
get_factors	7	0	6	0	100%
hanoi	5	0	2	0	100%
is_valid_parenthesization	6	0	6	0	100%
kheapsort	8	0	4	0	100%
knapsack	10	0	6	0	100%
kth	11	0	8	0	100%
lcs_length	8	0	6	0	100%
levenshtein	6	0	4	0	100%
lis	10	0	6	0	100%
longest_common_subsequence	6	0	4	0	100%
max_sublist_sum	7	0	2	0	100%
mergesort	19	0	6	0	100%
minimum_spanning_tree	11	0	6	0	100%
next_palindrome	14	0	6	0	100%

[4] https://github.com/RenataBrito/QuixBugs_report/blob/master/Results/pytest.pdf

next_permutation	9	0	8	1	88%
node	14	3	0	0	79%
pascal	10	0	4	0	100%
possible_change	7	0	4	0	100%
powerset	6	0	4	0	100%
quicksort	7	0	6	0	100%
reverse_linked_list	8	0	2	0	100%
rpn_eval	11	0	6	0	100%
shortest_path_length	29	0	6	0	100%
shortest_path_lengths	10	0	0	0	100%
shortest_paths	7	0	6	0	100%
shunting_yard	3	0	8	0	100%
sieve	13	0	5	0	100%
sqrt	6	0	2	0	100%
subsequences	5	0	6	0	100%
to_base	7	0	2	0	100%
topological_ordering	9	0	8	0	100%
wrap	7	0	2	0	100%
Total	373	3	193	1	99%

Tabela 3. Resultados para a execução da ferramenta *Pytest*

Apesar do *pytest* ser uma ferramenta estável e bastante popular no meio da comunidade *Python* [Snyk 2021], durante o desenvolvimento da atividade que compreende esse trabalho foram identificadas algumas dificuldades com relação à utilização da ferramenta. Tais dificuldades abrem espaço para possíveis melhorias futuras. Abaixo discutimos a respeito dos principais entraves encontradas e as soluções apontadas.

- **Problema de compatibilidade em plataformas que existem múltiplas versões do *Python* instaladas**
  O projeto foi feito inicialmente utilizando o sistema operacional Ubuntu 18.04.2 LTS, que possui uma instalação padrão do *Python* 2.7.15+. Contudo, o *pytest* recomenda a utilização da linguagem de programação *Python* 3.5 ou superior [Holger Krekel 2020].
  Ao executar os casos de teste observamos que internamente o *pytest* utiliza, por padrão, a versão mais antiga do *Python* disponível no sistema operacional (*Python* 2). Por padrão, o *pytest* não disponibiliza uma diretiva para especificar qual versão do *Python* deseja-se utilizar para executar os casos de testes.
  Devido a essa limitação, os casos de teste para programas que utilizem funcionalidades especificas das novas versões do *Python* podem apresentar comportamento inesperado. Tendo em vista esse problema, optou-se por migrar todo o projeto para execução em um ambiente virtual *Python* isolados utilizando ambientes virtualizados (*virtualenv*), que possibilitam controlar com precisão a versão das bibliotecas e da linguagem de programação utilizadas.
- **Problema para lidar com programas que possuem esquemas de módulos complexos**
  Boas práticas de programação orientam que projetos devem possuir uma estrutura de diretórios modularizada capaz de organizar funcionalidades distintas em diferentes diretórios.

Ao realizar uma busca na internet [5] é possível identificar que esse é um problema frequente, demonstrando que existem muitas dúvidas sobre como organizar a estrutura de diretórios de um projeto para que o *pytest* possa reconhecer corretamente os *imports* de programas e funções de forma correta, o que acaba tornando-se uma barreira para usuários iniciais.

Contudo, ao comparar com ferramentas mais maduras, entendemos que essa é uma solução que compromete o isolamento dos programas testados e entendemos que esse problema poderia ser lidado de uma forma mais simples pela própria ferramenta de testes.

6.2. Ferramentas sobre Teste de Mutação

As próximas avaliações estão relacionadas à aplicação do critério de teste de mutação (subseção 3.3). Assim como na subseção anterior, o objetivo dessa etapa consiste em avaliar o estado inicial do conjunto de casos de teste fornecido primariamente pelo *beachmark* utilizado nesse estudo e aprimorá-lo, quando possível, para que ao final do estudo possa atingir um critério de teste adequado.

6.2.1. *cosmic-ray*

Conforme descrito na subseção 5.2, a ferramenta *cosmic-ray* utiliza 14 operadores de mutação. A ferramenta foi capaz de gerar um total de 1802 mutantes para os 40 programas que compõem o *benchmark*.

Nós disponibilizamos no repositório do projeto os resultados da execução ferramenta utilizando o conjunto de casos de teste após adicionar novos casos de teste a fim de obter um conjunto de teste adequado para matar a maior parte dos mutantes gerados [6].

Por meio da realização dessa atividade foi possível observar as características e limitações da ferramenta *cosmic-ray*:

Dentre os aspectos negativos observados na utilização da ferramenta o mais crítico está relacionado ao fato de que a ferramenta apresenta um alto custo de processamento, consumindo todos os recursos disponíveis da máquina que realiza a sua execução. De acordo com a pesquisa realizada, essa é uma *issue #486*[7] já conhecida pelos desenvolvedores, mas ainda não há uma solução definitiva para a mesma.

Devido à limitação relacionada ao custo computacional de execução da ferramenta, representada com maiores detalhes na subseção 6.2.4, não foi possível investigar com maiores detalhes o estado dos mutantes sobreviventes para identificar se os mesmos tratavam-se de mutantes que poderiam ser mortos, ou se podem ser identificados como mutantes equivalentes.

---

[5] `https://bit.ly/2x7T7XV`

[6] `https://github.com/RenataBrito/QuixBugs_report/blob/master/Results/cosmicRay.pdf`

[7] `https://github.com/sixty-north/cosmic-ray/issues/486`

6.2.2. *mutmut*

Conforme descrito na subseção 5.2, a ferramenta *mutmut* utiliza 10 operadores de mutação. A ferramenta foi capaz de gerar um total de 486 mutantes para os 40 programas que compõem o *benchmark*.

Nós disponibilizamos no repositório do projeto uma tabela que apresenta os resultados da execução ferramenta utilizando o conjunto de casos de teste após o aprimoramento de novos casos de teste adicionados para obtenção de um conjunto de teste adequado ao critério teste de mutação para os mutantes gerados pela ferramenta [8].

Um ponto específico para os resultados dessa ferramenta é que os operadores de mutação disponíveis, em geral, exploram características genéricas a todas as linguagens de programação, deixando de lado aspectos específicos da linguagem de programação *Python*.

Outro aspecto relacionado aos resultados é o de que essa ferramenta foi a ferramenta que produziu um maior número de mutantes que foram mortos por *timeout*. Esse tipo específico de mutantes normalmente está relacionado a realização de alterações no código-fonte que impliquem na ocorrência de *loops* infinitos.

É importante ainda apontar com relação às características dessa ferramenta que durante a sessão de teste a mesma realiza as mutações em disco, ou seja, as mutações são realizadas em cima do código-fonte original do programa em teste e após a execução dos casos de teste a mutação é revertida para que o programa retorne ao seu estado original. Essa operação pode apresentar um risco em cenários que não façam uso de controle de versão, uma vez que se a ferramenta apresentar algum comportamento inesperado antes do final da sua execução, a mesma não será capaz de reverter a mudança inserida no código-fonte do programa em teste, podendo, acidentalmente, causar a inserção de defeitos no código fonte.

6.2.3. *mutpy*

Conforme descrito na subseção 5.2, a ferramenta *mutpy* possui um conjunto de 20 operadores de mutação convencionais e 7 operadores de mutação experimentais. Para a nossa avaliação nós consideramos os 20 operadores convencionais, e a ferramenta foi capaz de gerar um total de 483 mutantes para os 40 programas que compõem o *benchmark*.

No repositório do projeto nós apresentamos os resultados [9] para o conjunto de casos de teste final produzido neste trabalho. Ao observar o funcionamento e estabilidade das ferramentas, optou-se por investigar com maiores detalhes os mutantes produzidos pela ferramenta *mutpy*. Dessa forma, além dos casos de teste originalmente adicionados para atingir uma cobertura ideal para o critério ”todas-arestas”na ferramenta *pytest* e os casos de teste extras adicionados para as ferramentas de mutação *cosmic-ray* e *mutmut*, também foram acrescidos novos casos de teste para matar os mutantes que haviam inici-

[8] `https://github.com/RenataBrito/QuixBugs_report/blob/master/Results/mutmut.pdf`

[9] `https://github.com/RenataBrito/QuixBugs_report/blob/master/Results/mutpy.pdf`

almente sobrevivido nessa ferramenta.

Com relação às características da ferramenta, destaca-se que além do relatório específico ao teste de mutação a ferramenta também dá informações relacionadas à cobertura de comandos, facilitando a identificação do rastro de execução dos casos de teste para os mutantes gerados.

A análise dos mutantes sobreviventes e consequente adição de novos casos de teste pode ser observada em um arquivo descrito como *Apêndice* presente no repositório do projeto deste trabalho.

6.2.4. Custo computacional de execução das ferramentas de teste de mutação

Um dos fatores apontados como limitantes para adoção prática do teste de mutação em contextos industriais está relacionado ao seu alto custo computacional [Delamaro et al. 2021], que acaba se tornando um fator proibitivo, uma vez que essa tarefa passa a ser muito custosa e acaba desincentivando a utilização da técnica.

Com isso em mente, nós decidimos por avaliar o desempenho das ferramentas de teste de mutação investigadas nesse trabalho.

No repositório do projeto nós apresentamos uma tabela [10] que descreve o tempo de execução gasto em segundos para cada uma das ferramentas utilizadas na avaliação. Na Figura 3 é possível observar que, conforme citado na subseção 6.2.1, o custo computacional para execução da ferramenta *cosmic-ray* para os 40 programas que compõem o *benchmark* avaliado nesse trabalho é demasiadamente alto, quando comparado com relação às demais ferramentas.

A ferramenta que atinge o melhor custo computacional em relação ao número de mutantes gerados é a ferramenta *mutpy*. A ferramenta *mutmut* não apresentou um custo computacional proibitivo, mas uma vez que a sua estrategia de geração/execução de mutantes lida com demasiadas atividades de escrita no disco, seu desempenho acaba sendo prejudicado.

Como resultado final para essa análise, apontamos que o aspecto ligado ao custo computacional e gerenciamento dos recursos empregados na execução das ferramentas é um fator primordial para possibilitar que as mesmas ganhem popularidade e possam ter o seu habito de utilização cada vez mais popular em cenários de teste reais, sendo, portanto, uma das principais características a serem levadas em consideração no processo de aprimoramento de ferramentas para aplicação do teste de mutação para o contexto da linguagem de programação *Python*.

7. Conclusão

Os resultados da avaliação experimental desenvolvida durante o desenvolvimento trabalho permitiram observar o comportamento de ferramentas de teste de software para o contexto da linguagem de programação *Python*, bem como entender as suas principais capacidades e limitações de cada uma delas.

[10] `https://github.com/RenataBrito/QuixBugs_report/blob/master/Results/tempoDeExecucao.pdf`

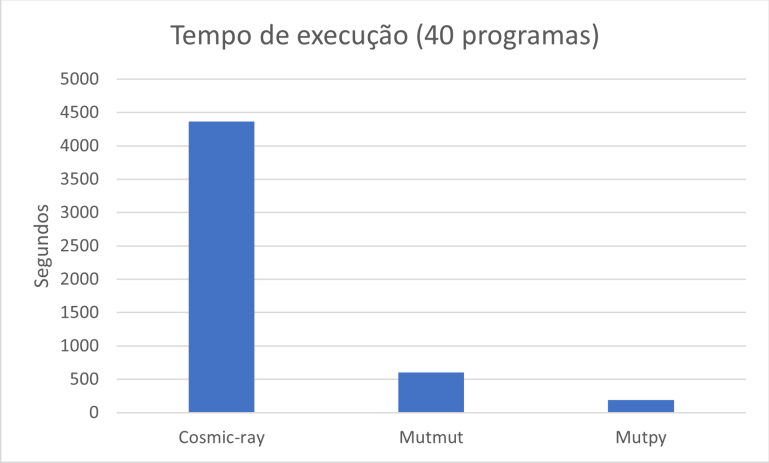

Figura 3. Tempo de execução dos casos de teste para as ferramentas de mutação para o experimento completo (41 programas).

Esse resultado é importante para dar subsídios teóricos e práticos no momento da escolha de ferramentas para o desenvolvimento de trabalhos no contexto de teste de software para a linguagem de programação *Python*, além de contribuir para a popularização e divulgação de resultados que alinham conceitos fundamentalmente desenvolvidos na academia aplicados em um contexto técnico, que pode ser absolvido de forma prática.

A avaliação experimental com relação ao custo computacional para aplicação do teste de mutação permitiu observar como as ferramentas investigadas se comportam em um contexto de estresse, no qual são exigidas para processar um grande volume de informações, de forma a simular um cenário de aplicação próximo de um cenário real a fim de apontar as principais limitações das mesmas.

Como principal contribuição desse trabalho, foi disponibilizado, de forma pública, um repositório[11] composto de um *benchmark* com 40 programas, desenvolvidos para a linguagem *Python*, contendo, para cada programa, um conjunto de casos de teste adequado aos critérios de teste estrutural (todas-arestas) e ao teste de mutação.

A partir do conjuntos de casos de testes adequados a diferentes critérios de testes, novos trabalhos, propostas de abordagens de teste, técnicas de geração automática de dados de testes, podem explorar os resultados deste trabalho, por meio de estudos comparativos, para atestar a eficiência/eficacia de tais abordagens em relação a critérios de testes clássicos cobertos neste trabalho.

Dessa forma, os resultados desse trabalho possibilitam que o *beanchmark* disponibilizado possa ser re-utilizado em futuros experimentos/investigações na área de engenharia de software experimental e no contexto de teste de software específico para o domínio da linguagem de programação *Python*.

Referências

Abingham, A. and Smallshire, R. (2021). *Cosmic Ray - mutation testing for Python*. `https://github.com/sixty-north/cosmic-ray`.

[11] `https://github.com/RenataBrito/QuixBugs_report`

Barbosa, E. F., Chaim, M. L., Vincenzi, A. M. R., Delamaro, M. E., Jino, M., and Maldonado, J. C. (2016). *Introdução ao Teste de Software – Capítulo 4 - Teste Estrutural*. Campus, Rio de Janeiro, 1 edition.

Delamaro, M. E., Andrade, S. A., de Souza, S. R. S., and de Souza, P. S. L. (2021). Parallel execution of programs as a support for mutation testing: A replication study. *International Journal of Software Engineering and Knowledge Engineering*, 31(03):337–380.

Delamaro, M. E., Maldonado, J. C., and Jino, M. (2016a). *Introdução ao Teste de Software – Capítulo 1 – Conceitos Básicos*. Campus, Rio de Janeiro, 2 edition.

Delamaro, M. E., Oliveira, R. A. P., Barbosa, E. F., and Maldonado, J. C. (2016b). *Introdução ao Teste de Software – Capítulo 5 – Teste de Mutação*. Campus, Rio de Janeiro, 2 edition.

Gilsing, V., Bekkers, R., Freitas, I. M. B., and Van der Steen, M. (2011). Differences in technology transfer between science-based and development-based industries: Transfer mechanisms and barriers. *Technovation*, 31(12):638–647.

Hałas, K. (2014). *MutPy - A Mutation Testing Tool for Python 3.x*. `https://github.com/mutpy/mutpy`.

Holger Krekel (2020). pytest documentation release 5.4. Available from: `https://buildmedia.readthedocs.org/media/pdf/pytest/latest/pytest.pdf`.

Hovmöller, A. (2021). *mutmut - python mutation tester*. `https://github.com/boxed/mutmut`.

Hynninen, T., Kasurinen, J., Knutas, A., and Taipale, O. (2018). Software testing: Survey of the industry practices. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1449–1454. IEEE.

Konrad Halas. Operators mutpy. Available from: `https://pypi.org/project/MutPy/`, year=2019.

Krekel, H. (2021). *pytest - helps you write better programs*. `https://docs.pytest.org/en/6.2.x/`.

Lin, D., Koppel, J., Chen, A., and Solar-Lezama, A. (2017). Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56.

Millman, K. J. and Aivazis, M. (2011). Python for scientists and engineers. *Computing in Science & Engineering*, 13(2):9–12.

Offutt, A. J. (1992). Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20.

Pressman, R. (2010). *Software engineering: a practitioner's approach*. McGraw-Hill higher education. McGraw-Hill Higher Education.

Sandler, C., Myers, G., and Badgett, T. (2012). *The Art of Software Testing*. John Wiley & Sons.

Sixty North AS. Operators cosmic ray. Available from: https://github.com/sixty-north/cosmic-ray/tree/master/src/cosmic_ray/operators, year=2019.

Snyk (2021). *pytest popularity on Snyk*. https://snyk.io/advisor/python/pytest.

Srinath, K. (2017). Python–the fastest growing programming language. *International Research Journal of Engineering and Technology*, 4(12):354–357.

The economist (2018). Python is becoming the worlds most popular coding language. Available from: https://www.economist.com/graphic-detail/2018/07/26/python-is-becoming-the-worlds-most-popular-coding-language.

Tiobe the software quality company (2021). Tiobe index for january 2021. Available from: https://www.tiobe.com/tiobe-index/.

Van Rossum, G. et al. (2007). Python programming language. In *USENIX annual technical conference*, volume 41, page 36.