Universidade de São Paulo (USP)
Instituto de Ciências Matemáticas e de Computação (ICMC)
Departamento de Sistemas de Computação (SSC)

Bolsista – Renata Oliveira Brito (renata.oliveira.brito@usp.br)
Orientador – Prof. Dr. Márcio Eduardo Delamaro (delamaro@icmc.usp.br)
co-Orientador – Me. Stevão Alves de Andrade (stevao@icmc.usp.br)

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

Relatório final apresentado à USP (Universidade de São Paulo) com o objetivo de apresentar as atividades realizadas pela aluna de iniciação cientifica Renata Oliveira Brito. O relatório compreende ao período entre os meses de 09 de abril de 2020 a 08 de outubro de 2020.

São Carlos – SP
08 de outubro de 2020

Sumário

1. Plano de Trabalho

Esta seção apresenta o plano de trabalho definido no projeto inicial de maneira sumarizada. Também são apresentadas as atividades propostas no projeto de doutorado.

1.1. Resumo do Projeto Proposto

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, sua sintaxe direta e uso obrigatório de identação de código, que facilita o aprendizado e a leitura de código, possibilitando uma rápida curva de aprendizado, além de ter popularizado a utilização de novas tecnologias de computação, como, por exemplo, aprendizado de máquina. Entretanto a popularização de novas linguagens de programação também deve ser acompanhada pela popularização de ferramentas e técnicas que visem garantir a qualidade no processo de construção de software. Considerando esse cenário, julga-se necessário realizar a avaliação de técnicas e ferramentas a fim de medir a adequação delas no contexto dessas novas tecnologias que tem se popularizado nos últimos anos. Esse projeto visa colaborar com essa problemática por meio da categorização e avaliação de técnicas e critérios de teste de software aplicados a linguagem de programação *Python*. Planeja-se revisar os conceitos teóricos referentes aos principais critérios das técnicas de teste de software estrutural e teste de mutação e avaliar como tais abordagens tem sido utilizado no contexto da linguagem de programação *Python*. Entre as atividades deste projeto, espera-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 das mesmas no contexto de projetos *open-source*.

1.2. Atividades Planejadas

As principais atividades planejadas para desenvolvimento desta pesquisa são:

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.
2. **Revisão bibliográfica sobre conceitos de Teste de Software**: após a conclusão da primeira atividade serão revisados os 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 da linguagem de programação *Python*.
3. **Catalogar ferramentas de teste de software em *Python***: essa etapa corresponde ao levantamento de ferramentas/tecnologias que podem dar suporte a aplicação de teste de software utilizando a linguagem de programação *Python*.
4. **Catalogar projetos em *Python* disponíveis em repositórios *open-source***: serão catalogados alguns 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, a aluna será encorajada a utilizar as ferramentas e catalogar os resultados referentes a pratica de teste de software nos repositórios analisados.
6. **Criação de um benchmark de programas e conjuntos de casos de teste adequados:** com a consolidação dos resultados dos objetivos anteriores, planeja-se utilizar os resultados para construção de um benchmark que pode ser utilização em experimentos futuros utilizando a linguagem de programação *Python*.

As demais seções desse relatório compreendem a uma discussão individual de cada uma das atividades propostas, enumeradas acima, e estão organizadas da seguinte forma: A seção 2 apresenta os resultados sobre o estudo da linguagem de programação *Python*, a seção 3 apresenta os resultados da revisão bibliográfica sobre os temas de teste de software investigados durante esse projeto de iniciação cientifica, a seção 5 discute as ferramentas de teste de software selecionadas para o desenvolvimento desse projeto, a seção 4 apresenta informações sobre o *benchmark* utilizado nas atividades, 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 desse projeto de iniciação cientifica.

Reforça-se que todos os artefatos desenvolvidos durante o período dessa iniciação cientificam encontram-se disponíveis publicamente no repositório de código *open-source* `https://github.com/RenataBrito/quickbugs_report/`.

2. Revisão bibliográfica sobre conceitos da linguagem de programação *Python*

Partindo de um projeto de natal e por aborrecimentos pelas deficiências em outras linguagens de programação, em 1989 Guido van Rossum[1], um matemático e cientista da computação holandês, formado pela Universidade de Amsterdã, criou a linguagem de programação *Python*.

Como vemos no gráfico apresentado na Figura 1, a linguagem de programação *Python*, 30 anos depois de sua criação, vem crescendo cada vez mais em comparação às demais. O número de busca 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.

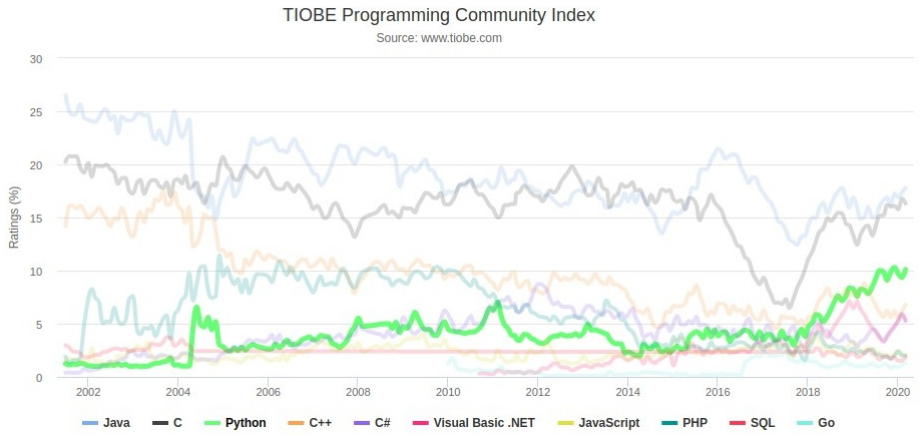


Figura 1. Ranking com as linguagens de programação mais populares de 2020 segundo a TIOBE

[Tiobe the software quality company 2020]

As duas principais características responsáveis pela popularização dessa linguagem são a sua simplicidade, com uma sintaxe enxuta e direta em conjunto com a obrigatoriedade do uso de identação de código, e sua flexibilidade, abrangendo ferramentas que possibilitem o desenvolvimento de projetos desde *machine learning* até desenvolvimento de jogos e estudos sobre dados científicos [Guido van Rossum 2019]. Os "pythonistas", como são conhecidos os aficionados pela linguagem, ajudaram adicionando mais de 145.000 pacotes que podem servir de auxílio no desenvolvimento dos mais variados tipos de projetos.

Segundo o Codecademy [Codecademy education company 2020], um site que possuí uma plataforma interativa online que oferece aulas gratuitas de programação em linguagens, e que já ensinou mais de 45 milhões de novatos a usar as mais variadas linguagens de programação, o maior aumento na demanda é daqueles que desejam aprender *Python*.

Segundo o *survey* anual (referente ao ano de 2019) do maior sites de perguntas e respostas de tópicos ligados a ciência de computação e linguagens de programação em geral, *Stack Overflow* [StackOverflow 2019], um dos principais resultados foi com relação

[1] `https://pt.wikipedia.org/wiki/Guido_van_Rossum`

à linguagem *Python*. Os resultados apontaram que a linguagem se tornou a linguagem de programação que mais cresce, subindo novamente no ranking das linguagens de programação na pesquisa do site, superando o *Java* este ano e permanecendo como a segunda linguagem mais amada (atrás apenas de *Rust*). *Python* é a linguagem mais procurada pelo terceiro ano consecutivo. Os resultados do *survey* apontam ainda que os desenvolvedores que ainda não a utilizam a linguagem, apontaram que é uma linguagem que desejam aprender, direcionando dessa forma para uma tendência de crescimento nos próximos anos.

O estudo teve como principal finalidade solidificar os conhecimentos da aluna com relação às características da linguagem de programação *Python*, uma vez que, para o cumprimento das atividades previstas no trabalho, é necessário a análise de códigos fonte que utilizam os conceitos existentes nessa linguagem de programação. Dessa forma, essa etapa consistiu em estudos dirigidos, por meio de aplicação prática, abordando os principais conceitos da linguagem, tais como:

- variáveis;
- comandos de seleção, if/elif/else;
- comandos de repetição for/while/continue + break;
- listas/tuplas/matrizes;
- funções/recursão;
- operações com arquivos;
- tratamento de exceções;
- criação de ambientes virtuais.

3. Revisão bibliográfica sobre conceitos de Teste de Software

A tecnologia está crescendo cada vez mais, e com isso a demanda de software tem aumentado, testar esses softwares é talvez a tarefa mais cara de um projeto, além do custo dos profissionais temos o custo que envolvem os softwares, visto que softwares com mal funcionamento causam grandes despesas [Delamaro et al. 2016], por mais que planeje toda sua estrutura, existe sempre a possibilidade da abertura de processos, o deixando com falhas segundo sua documentação, se tornando um software inconfiável. E quem faz esses softwares? São os seres humanos, que mesmo com todo os estudos de teorias e horas de prática são passíveis de vários erros, desde sua habilidade até sua interpretação. Para que tais acontecimentos prejudiciais ao software sejam descobertos e tratados, existe uma série de atividades, coletivamente chamadas de "Validação verificação e teste"ou "VV&T", com a finalidade de minimizar as falhas a ponto de não existir mais falhas na documentação.

O teste de software é uma atividade dinâmica, seu intuito é executar o programa utilizando algumas entradas específicas e identificar se seu comportamento está de acordo com sua especificação ou o seu esperado. Caso a execução apresente resultados não especificados, dizemos que foi identificado um erro ou um defeito, porem essa atividade é um tanto quanto complexa, pois os erros não têm uma ordem de acontecer nem uma regra, por isso a atividade de teste é dividida em fases com objetivos distintos. No geral, podemos dizer que existe três fases de teste, a primeira como teste de unidade, a segunda o teste de integração e a terceira o teste de sistemas, onde cada um tem sua estrutura e é fundamental suas execuções.

Portanto um grande objetivo é mostrar a presença de defeitos caso eles existam, melhorando a qualidade do software, testando-o juntamente com eficácia nos processos, podendo se tornar um grande passo para reduzir custos de software no geral.

3.1. Teste Estrutural

O Teste estrutural também conhecido como teste de caixa branca, está relacionado com o teste das menores unidades de código de um programa. Com isso tem como objetivo testar o código por completo, verificando linha a linha o comportamento de seus módulos e fluxos.

A arquitetura dos testes é única para cada tipo de programa a ser testado e projetada a partir da estrutura interna individualmente, sendo necessário conhecer o código fonte e entender o seu funcionamento. Com essa atividade o testador busca ter a certeza de que toda estrutura do programa tenha sido devidamente exercitada. Um critério para definir a qualidade de um conjunto de teste para essa técnica é avaliar o quanto os testes são capazes de cobrir a maior parte do código fonte do programa analisado [Barbosa et al. 2016].

Vale ressaltar também que o teste estrutural também possui algumas desvantagens, porém é uma técnica complementar às demais sendo seus resultados relevantes para a manutenção e evolução do ciclo de desenvolvimento do software [Pressman and Maxim 2016], possibilitando obter cada vez mais avaliações a respeito da confiabilidade do software.

O grafo de fluxo de controle (GFC) ou grafo de programa, serve para que possamos fazer uma abstração da estrutura do programa. Esse grafo é criado a partir de blocos disjuntos de comandos que compõem um programa, no qual um bloco de comandos é

representado pelos vértices do grafo, onde a execução do primeiro comando do bloco faz com que todos os outros comandos desse bloco sejam executados sequencialmente e sem desvios. Quando um bloco termina de ser executado um novo bloco inicia a sua execução, criando-se assim uma aresta no grafo que aponte para o próximo bloco de código representado pelo vértice seguinte.

Exemplo do GFC para uma função de um programa que representa a operação para identificar o máximo divisor comum entre dois números:

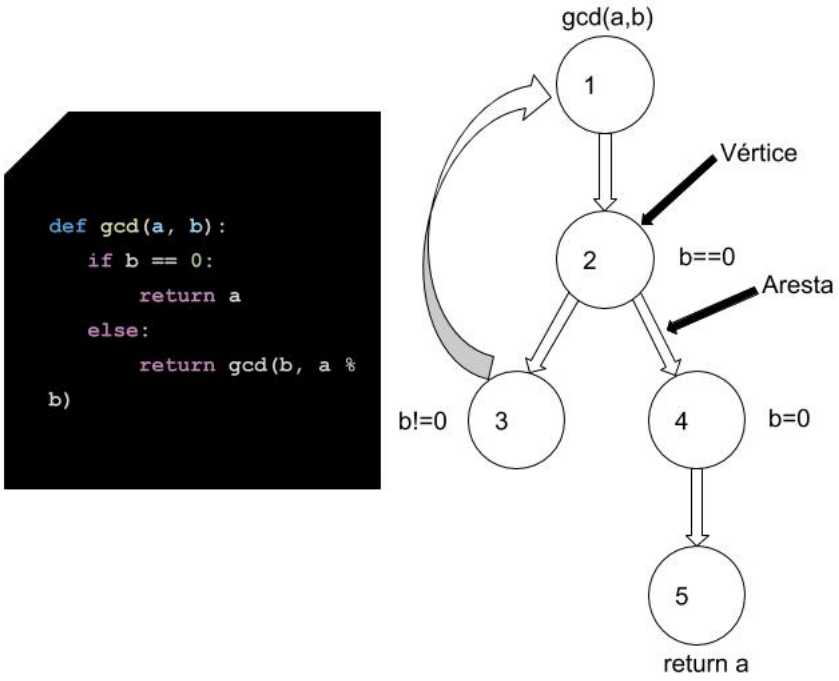


Figura 2. Grafo de fluxo de controle de um programa estudado, gcd

Podemos aplicar os critérios de testes mais populares do teste estrutural, "Todos-nós"ou "Todos-arcos". O critério Todos-nós significa que devemos criar casos de teste que faça com que cada instrução que compõe o GFC seja executada pelo menos uma vez pelo conjunto de todos os casos de teste, fazendo com que ao final da sessão de testes todos os vértices tenham sido percorridos.

Já o critério Todos-arcos, significa que devemos criar um conjunto de casos de teste que seja capaz de percorrer pelo menos uma vez todas as arestas que compõem o GFC. Ou seja executar todos os desvios que representam o fluxo/arestas de execução do nosso programa [Barbosa et al. 2016].

3.2. Teste Baseado em Defeitos - Teste de Mutação

Essa técnica de teste complementa os demais critérios das técnicas de teste funcional e estrutural. 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. A teoria por trás do teste de mutação atesta que casos de teste capazes de identificar erros ocasionados a partir de defeitos simples são tão sensíveis que, implicitamente, também são capazes de revelar

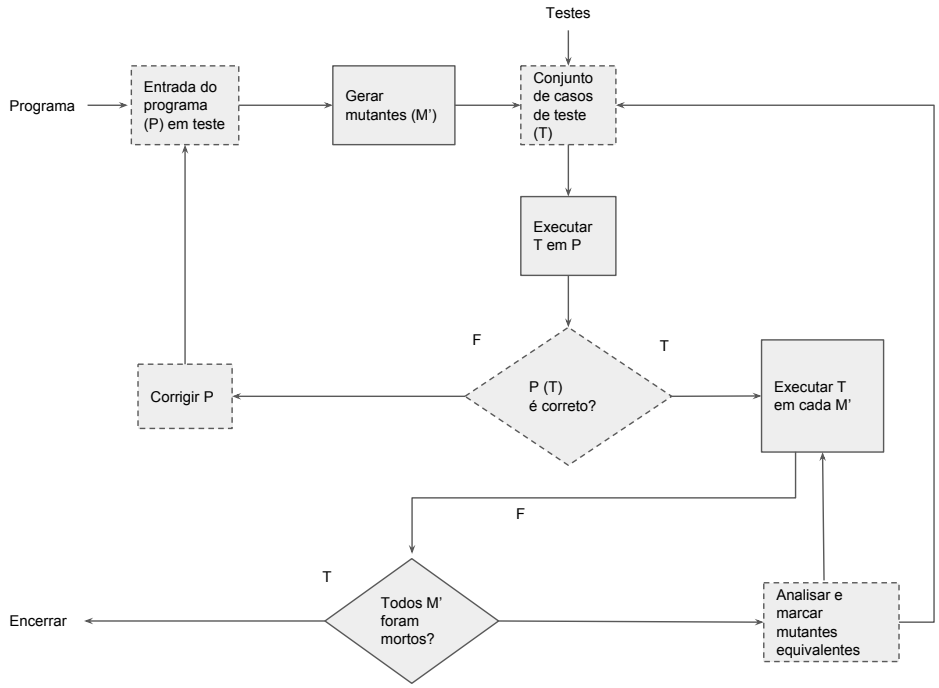


Figura 3. Processo genérico do Teste de Mutação. Fonte: Adaptado de [Offutt and Untch 2001]

outros tipos de erros, devido a uma característica conhecida como efeito de acoplamento, que foi evidenciada por meio de experimentos científicos [Offutt 1992].

Para aplicarmos o critério de teste de mutação é necessário realizar as seguintes atividades:

- Gerar os mutantes
- Executa o programa
- Executa os mutantes
- Análise dos mutantes

O processo geral envolvido na condução das atividades do Teste de Mutação é representado pela Figura 3. Destaca-se que as atividades destacadas por caixas tracejadas são, normalmente, tarefas desempenhadas manualmente. O processo é, portanto, normalmente, divido nas seguintes etapas:

A aplicação do critério se encerra quando todos os mutantes que foram gerados, são mortos pelo conjunto de casos de teste, ou há a identificação de mutantes que não podem ser mortos, no caso da existência de mutantes equivalentes.

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 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). Dessa forma os operadores de mutação podem variar de acordo com a linguagem de programação do programa, um exemplo é a linguagem *C*, que possui um conjunto de 75

operadores de mutação, gerados a partir de pequenas variações em estruturas de código como comandos, variáveis, contantes e operadores relacionais (>,<,>=, etc).

Abaixo especificamos alguns exemplos para cada tipo:

- Mutação de comando:
  - **SWDD**, Troca um comando while por do-while
- Mutação de Operadores:
  - **ORRN**, Troca operador relacional por outro operador relacional
- Mutação de variáveis:
  - **VTWD**, Troca referência escalar pelo sucessor e predecessor
- Mutação de constantes:
  - **CCSR**, Troca referências escalares por constantes

Há uma medida para avaliar a qualidade da atividade de teste de mutação, denominada "escore de mutação" (*mutation score*), que pode variar entre 0 e 1 (100%). O escore de mutação é dado pela fórmula descrita abaixo:

Escore de mutação = mutantes mortos / total de mutantes - mutantes equivalentes.

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 sintática 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 equivalente.

Um mutante é equivalente possui a mesma semântica do programa original, porém os códigos são sintaticamente diferentes. A etapa da análise dos mutantes equivalentes é fundamental para o correto cálculo do escore de mutação e pode se tornar um dos principais obstáculos da adoção prática do critério. Pois com base no cálculo do escore de mutação os testadores podem avaliar a qualidade de um dado conjunto de casos de teste.

Vemos então a limitação nesse critério, o qual já possui um alto custo de aplicação, pois mesmo em um programa relativamente simples há um número grande de mutantes que podem ser gerados e precisam ser executados. Visto essa limitação, existem diversos trabalhos na literatura que abordam alternativas como mutação aleatória, mutação seletiva e mutação restrita, nas quais cada abordagem tem como objetivo viabilizar a aplicação do critério de mutação em ambientes reais de desenvolvimento de software.

Concluímos que a utilização desse critério é uma ótima ferramenta para criar conjuntos de casos de teste que sejam capazes de revelar defeitos no software testado, para isso é necessário avaliar todo o conjunto de mutantes gerados, tendo como objetivo encontrar um conjunto de teste que seja capaz de matar todos os mutante que foram criados, pois quanto maior o escore de mutação mais adequado é o conjunto de casos de teste para o programa em teste e, por tanto, maior a confiança a respeito da sua confiabilidade.

4. Catalogar projetos em *Python* disponíveis em repositórios *open-source*

Uma das etapas desse projeto corresponde a catalogar projetos de software disponíveis em repositórios *open-source*, com intuito de atestar a aplicabilidade das ferramentas discutidas na subseção anterior.

Nós optamos por utilizar um conjunto de programas composto por 40 programas traduzidos para *Python*, cada programa contendo apenas um único bug em uma única linha. O conjunto de programas que compõe esse *benchmark* é chamado de *QuixBugs* [Lin et al. 2017] 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.

A vantagem de utilizar esse repositório é o fato de que de antemão nós temos acesso a um *benchmark* representativo de programas que já possui um conjunto mínimo de casos de teste que revela os defeitos existentes nos programas. Além do fato do *benchmark* possuir tanto as versões dos programas com defeitos quanto a versão corrigida, possibilitando que analises experimentais mais profundas possam ser realizadas.

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

Tabela 1. 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

4.1. Descrição dos programas contidos no *benchmark*

Parte das atividades desse trabalho consistia no entendimento dos programas contidos nesse *benchmark* a fim de possibilitar o desenvolvimento dos demais objetivos. Como resultado dessa atividade abaixo apresentamos uma breve descrição para cada um dos programas contidos no *benchmark*:

- `bitcount`: retorna a quantidade de números 1 de um determinado número em sua representação binária;
- `breadth first search`: do português, busca em largura, começa pelo vértice raiz e explora todos os vértices vizinhos. Então, para cada um desses vértices mais próximos, explora-se os seus vértices vizinhos inexplorados e assim por diante, até que ele encontre o alvo da busca;

- `bucketsort`: ordena um *array* dividindo-o em um número finito de recipientes (k = passando como parâmetro como o maior número possível no *array*) cria os k *buckets*, adiciona 1 em cada valor dentro do array exemplo: k=4, valor=3 = [0,0,1,0] enumera o vetor que contém 0 ou 1 [0,0,1,0] = [(0,0),(1,0),(2,1),(3,0)] e adiciona ordenado em um *array* multiplicando um iterador para cada posição, montando assim o vetor ordenado;
- `depth first search`: do português, busca em profundidade, começa num nó raiz (*startnode*) e vai se aprofundando cada vez mais, até que o dado (*goalnode*) de busca seja encontrado ou até que ele se depare com um nó que não possui filhos (*leafnode*). Então a busca retrocede e começa no próximo nó;
- `detect cycle`: retorna verdadeiro se o grafo direcionado contém pelo menos um ciclo e falso caso não;
- `find first in sorted`: retorna a primeira aparição de um determinado valor em um *array*, caso não exista retorna -1, utilizada busca binaria;
- `find in sorted`: encontra por busca binária a posição de um determinado valor;
- `flatten`: retorna um único *array*, com um elemento cada posição. Caso tenha uma lista no *array* enviado, essa lista é particionada e cada elemento dessa lista é adicionado no vetor final a partir da sua posição inicial;
- `gcd`: o máximo divisor comum entre dois ou mais números reais é o maior número real que é fator de tais números;
- `get factors`: retorna um vetor com os valores da fatoração do número;
- `hanoi`: com 3 pilastras, inicialmente *start* em 1 e *end* em 3. É necessário mover todos os discos para o pino da direita Deve mover um disco de cada vez, sendo que um disco maior nunca pode ficar em cima de um disco menor. O programa recebe como parâmetro o número de discos, em qual pilastra começa e em qual devemos colocar, como padrão começa na mais esquerda 1 e deve terminar na mais a direita 3. Ele retorna as movimentações que devemos fazer (x,n) da pilastra x para pilastra n;
- `is valid parenthesization`: verifica se os parênteses estão dispostos de maneira correta, isto é, sempre quando aberto "(" deve, obrigatoriamente, que existir o que fecha ")";
- `kheapsort`: algoritmo de ordenação, com *heap*. Se P é um nó pai de C, então a chave (o valor) de P é maior que ou igual a (em uma *heap* máxima) ou menor que ou igual a (em uma *heap* mínima) chave de C;
- `knapsack`: problema da mochila. Solicita como parâmetro a capacidade que a mochila consegue carregar e a lista dos itens contendo cada item (a,b), a = peso, b = valor;
- `kth`: solicita como parâmetro um vetor, e um número k, onde k deve ser menor que o tamanho do array, e então encontra qual o valor k está o valor ordenado;
- `lcs length`: retorna o comprimento da maior subsequência presente em ambas as *strings* analisadas;
- `levenshtein`: encontra o número mínimo de operações necessárias para transformar uma *string* em outra;
- `lis`: retorna o comprimento da subsequência mais longa de um *array* no qual seus elementos são dispostos em ordem crescente;
- `longest common subsequence`: encontra qual a maior sequência dos elemento em duas strings;
- `max sublist sum`: retorna a maior soma de n números consecutivos do *array*;

- `mergesort`: algoritmo de ordenação que divide o problema em vários subproblemas e resolve esses subproblemas através da recursividade, após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas;
- `minimum spanning tree`: *spanning tree*, é a arvore formada por um grafo não direcionado que abrange todos os vértices. Então a *minimum spanning tree*, é o mínimo "caminho"do grafo que abrange todos os vértices;
- `next palindrome`: um número inteiro positivo é denominado palíndromo se sua representação no sistema decimal for a mesma quando lido da esquerda para a direita e da direita para a esquerda. Dado um número palíndromo, a função retorna o próximo número palíndromo;
- `next permutation`: reorganiza os números na próxima maior permutação de números lexicograficamente. Se tal arranjo não for possível, ele reorganizará na ordem mais baixa possível (ou seja, classificado em ordem crescente);
- `pascal`: o triângulo de Pascal é um triângulo numérico infinito formado por números binomiais (n, k) onde *n* representa o número da linha e *k* representa o número da coluna, iniciando a contagem a partir do zero. O programa retorna uma lista com cada linha, começando com a primeira;
- `possible change`: o programa retornar quantas maneiras podemos fazer mudança no *array* informado para termos um determinado valor informado;
- `powerset`: o conjunto de todos os subconjuntos de um conjunto dado A é chamado de conjunto de partes (ou conjunto potência) de A, denotado por P(A) ou 2 A 2**A;
- `quicksort`: algoritmo de ordenação, um elemento é selecionado como o pivô e a lista é dividida de forma que todos os elementos antes do pivô sejam menores do que ele e todos os elementos após o pivô sejam maiores do que ele. No final do processo, o pivô estará em sua posição final, e haverá duas sulistas não ordenadas. Essa operação é chamada de particionamento. Recursivamente é ordenando as sub lista dos elementos menores e maiores;
- `reverse linked list`: dado o nó principal de uma *linked list*, a programa reverte a *linked list*. Exemplo: input = 1->2->3->NULL output: 3->2->1->NULL;
- `rpn eva`: avalia expressões em notação polonesa reversa;
- `shortest path length`: é dado um gráfico ponderado, um vértice destino e um vértice origem, o programa retorna o menor caminho da origem até o destino;
- `shortest path lengths`: é dado um grafo ponderado e o número total de seus vértices, o programa retorna um dicionário com todos os possíveis caminhos no grafo juntamente com o seus respectivos pesos;
- `shortest paths`: dado um grafo orientado e ponderado, e um nó, o programa retorna um dicionario com todos os vértices e os pesos deles referente ao nó enviado;
- `shunting yard`: o algoritmo recebe uma expressão matemática especificada em notação de infixo. Ele retorna uma *string* de notação pós-fixada, também conhecida como notação polonesa reversa (RPN);
- `sieve`: dado um número *n*, imprime todos os primos menores ou iguais a *n*. Também é dado que n é um número pequeno. Retorna um *array* com os números primos;
- `sqrt`: dado um *epsilon* o programa retorna a raiz quadrada, ou aproximada se não exata, do número *epsilon* com uma “margem de erro”;

- `subsequences`: dado um valor no formato (*a*, *b*, *k*) o programa retornará um *array* com a quantidade máxima de subsequências que é possível realizar com valores de *a* - *b*-1 no tamanho de *k*;
- `to base`: dado um número em decimal *A* e uma base *B*, o programa resulta na representação do número decimal *A* na base *B*;
- `topological ordering`: A ordenação topológica para Grafo Acíclico Direcionado (DAG) é uma ordenação linear de vértices tal que para cada aresta direcionada (*u*, *v*), o vértice *u* vem antes de *v* na ordenação. Cada DAG pode ter uma ou mais ordenações topológicas;
- `wrap`: dado parágrafo em texto (uma *string*) retorna uma lista na qual cada elemento representa uma linha que tenha, no máximo, o comprimento de caracteres de largura informado por parâmetro.

5. Catalogar ferramentas de teste de software em *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*. Para o primeiro relatório, foram catalogadas as ferramentas relacionadas à aplicação do critério de teste de software estrutural.

As duas principais tecnologias mantidas para dar suporte à aplicação de teste estrutural são *Unittest* e *pytest* e são descritas com maiores detalhes nas subseções abaixo:

5.1. *unittest*

O *framework* de teste *Unittest* foi originalmente inspirado pelo popular *framework* de testes para a linguagem de programação *Java*, *JUnit* e tem diversas similaridades aos principais *frameworks* de teste de unidade em outras linguagens de programação.

Ele suporta automação de testes, compartilhamento de códigos de configuração, habilitação/desabilitação de testes, agregação de testes em conjuntos de casos de teste e independência dos testes da estrutura de relatórios.

Dentre os principais pontos positivos desse *framework* de testes é possível destacar:

- É parte da biblioteca padrão do *Python*. O *unittest* faz parte da biblioteca padrão do *Python* desde a versão 2.1. Isso o torna amplamente disponível para os desenvolvedores, sem a necessidade de instalar módulos adicionais, uma vez que ele vem instalado por padrão junto com a linguagem de programação.
- Sua adoção é mais simples para pessoas que estão se adaptando à linguagem de programação *Python*, uma vez que o *unittest* foi derivado do *JUnit*, possuindo então uma natural similaridade com outros *frameworks* da família *xUnit*.
- Um ponto importante é que o *unittest* utiliza uma serie de conceitos relacionados ao paradigma de programação orientada a objetos, o que acaba sendo um fator negativo para a nossa avaliação, uma vez que o *benchmark* de programas utilizado nesse trabalho possui em sua maioria programas implementados utilizando o paradigma estrutural.

Portanto, para os fins do desenvolvimento do projeto, nós optamos por aprofundar na utilização da tecnologia *pytest*, uma vez que ela melhor se adequa às características dos programas utilizados.

5.2. *pytest*

Podendo ser utilizado em *Python* 2 e *Python* 3, *pytest* 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 foi escrito, 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.

O *pytest* pode se destacar em comparação a outras estruturas de teste, pelo fato de que, seus testes são fáceis de ler, utiliza-se apenas o *assert* para a verificação de um teste e o *pytest* também executa testes escrito em *Unittest* ou *Nose*.

Com o *pytest*, os desenvolvedores têm uma maior liberdade na criação de testes, basta iniciá-los com "`test_`", sendo capaz de lidar com problemas cada vez mais envolvente. Um ponto negativo é que quando escrito o teste não poderá usá-los com nenhuma outra estrutura de teste, tendo que reescrever o teste praticamente inteiro.

O *plug-in pytest-cov* fornece um conjunto mínimo limpo de opções de linha de comando que são adicionadas ao *pytest*, juntamente com um suporte de cobertura para mostrar quais linhas de código foram testadas e quais não foram. Também inclui a porcentagem de cobertura de teste.

Instalado o *pytest-cov*, é possível executar *pytest* com a opção `-cov`. Se não for especificado nenhum parâmetro extra, a ferramenta disponibilizará um relatório de cobertura contendo informações de todas as bibliotecas *Python* utilizadas no programa alvo. Por isso é interessante que forneça um argumento para –cov especificar quais programas você deseja testar, como por exemplo `-cov=path, -cov-report=type` entre outros [Holger Krekel 2020]. A ferramenta ainda possibilita obter o relatórios da cobertura em outros formatos como html, tornando mais fácil a visualização.

Dessa forma, utilizando o *pytest* e o *plug-in pytest-cov* para criar testes de software é possível ter uma boa visualização da cobertura de uma forma visual, que possibilita um processo de tomada de decisão mais claro e otimizado pelo responsável de uma equipe de testes.

Dentre as principais características existentes que motivam a utilização do *pytest*, é possível destacar:

- permite a criação de suítes de teste compactas: as características que o *pytest* introduziu permitiram, pela primeira vez, uma mudança na comunidade *Python*, porque tornaram possível que os conjuntos de casos de testes fossem escritos em um estilo muito compacto em comparação ao *unittest*, o que vai de encontro às características da linguagem *Python*, que preza pela simplicidade na forma de escrever o código fonte;
- o *pytest* introduziu o conceito de que os testes do *Python* devem ser funções simples do *Python*, em vez de forçar os desenvolvedores a incluírem seus testes em grandes classes de teste como preconizado pelo *unittest*;
- informações sobre falhas de execução, mas claras e úteis: com a maioria das outras ferramentas, é necessário utilizar um depurador ou o *log* extra para descobrir de onde veio algum valor na execução do teste. O *pytest* reescreve os testes para que ele possa armazenar todos os valores intermediários, ocorridos durante a execução do programa, que podem levar à falha do teste e fornece uma explicação muito detalhada sobre o funcionamento do programa ao falhar a execução de algum caso de teste;
- mínimo de convenções de código: Os testes escritos com *pytest* precisam de pouquíssimo código padronizado, o que os torna fáceis de escrever e entender;
- *fixtures* são simples e fáceis de usar: uma *fixture* é apenas uma função que retorna um valor e, para usá-la, basta adicionar um argumento à função de teste;
- também é possível parametrizar diferentes testes, para que eles sejam executados com um conjunto de valores especificados por parâmetros, sem necessidade de reescrever o teste para cada dado de teste diferente;
- muitos *plugins* disponíveis: o *pytest* pode ser facilmente estendido com vários *plugins*, e a mesma equipe desenvolve vários *plugins* muito úteis. Por exemplo, o *pytest-cov* que possibilita verificar relatórios de cobertura para auxiliar a tomada de decisão do testador.

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 nas subseções abaixo:

5.3. *cosmic-ray*

Desenvolvido por Austin Abingham e Robert Smallshire, 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, que são descritos abaixo:

- *binary operator replacement*
- *boolean replacer*
- *break continue*
- *comparison operator replacement*
- *exception replacer*
- *keyword replacer*
- *no op*
- *number replacer*
- *operator*
- *provider*
- *remove decorator*
- *unary operator replacement*
- *util*
- *zero iteration for loop*

A ferramenta implementa os opradores de mutação por meio de *plug-ins*, para possibilitar aos usuários estender o conjunto de operadores disponível, fornecendo seus próprios operadores.

Para executar uma sessão de teste de mutação a ferramenta necessita que especifique um arquivo de configuração *TOML*, que especificará uma serie de parâmetros de configuração do código-fonte a ser testado. Além de especificar informações como os módulos que deseja realizar as mutações, a ferramenta de teste responsável por fazer a execução dos casos de teste (no nosso caso, *pytest*), os *scripts* de teste a serem utilizados, além de uma extensa lista de demais configurações.

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.

O *cosmic-ray* ainda adota a abordagem simples de usar um tempo limite para determinar quando eliminar um determinado caso de teste e considerá-lo incompetente. Para tal, basta especificar no arquivo de configuração *TOML* o tempo que considera satisfatório.

5.4. *mutmut*

Desenvolvido por Anders Hovmöller 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.

Dentre os principais objetivos focados na implementação da ferramenta, destaca-se:

- Possibilidade de lidar com mutantes e realizar operações de forma específica, tratando-os por um índice de identificação, não necessitando realizar a execução de todos os mutantes a cada iteração na etapa de testes.
- Execução por meio de linha de comando, possibilitando também salvar mutantes específicos no disco rígido para realizar análise individual.

Com a ideia de ser de fácil o uso, depois de realizada a instalação, basta digitar *mutmut run* no terminal. Isto executará por padrão o *pytest* ou o *unittest*, caso o *pytest* não estiver disponível.

A ferramenta implementa uma lista de 10 operadores de mutação, descritos abaixo:

- *operator*
- *keyword*
- *number*
- *name*
- *or test*
- *and test*
- *lambdef*
- *expr stmt*
- *decorator*
- *annassign*

Como a ferramenta busca descobrir onde o código-fonte que deverá ser mutado está, é possível que ocorram algumas inconsistências, isto é, caso existam trechos de código em formato de comentário, ou algumas funções além da qual se deseja gerar os mutante, a ferramenta considera tais trechos de código como parte para avaliação para geração dos mutantes. Então, uma situação especifica da ferramenta, é que para extrair o melhor resultado, é necessário executar a função interessada separada do restante do código do projeto.

Um ponto importante é que a execução tende a ser lenta, uma vez que a mutação é realizada em disco, em tempo de execução e não é paralelizada, tendo que executar todo o conjunto de testes que foi especificado para cada mutação.

5.5. *mutpy*

Desenvolvido por Konrad Halas, a ferramenta de teste de mutação *Mutpy* suporta código-fonte *Python* 3.3+.

Após a instalação da ferramenta, é necessário que todos os arquivos que serão utilizados na sessão de teste estejam no mesmo diretório. Para a execução é necessário especificar o modulo do programa que se deseja testar, o sinalizador para apontar para os módulos (`–target` e `–unit–test`), e o argumento que especifica a biblioteca que será utilizada para executar os testes `–runner pytest –coverage`.

A ferramenta suporta as bibliotecas *pytest* e *unittest* para execução dos casos de teste.

A ferramenta possui 20 operadores de mutação padrões e mais uma lista de 7 operadores de mutação experimentais. Abaixo são descritos os operadores suportados:

- *AOD - arithmetic operator deletion*
- *AOR - arithmetic operator replacement*
- *ASR - assignment operator replacement*
- *BCR - break continue replacement*
- *COD - conditional operator deletion*
- *COI - conditional operator insertion*
- *CRP - constant replacement*
- *DDL - decorator deletion*
- *EHD - exception handler deletion*
- *EXS - exception swallowing*
- *IHD - hiding variable deletion*
- *IOD - overriding method deletion*
- *IOP - overridden method calling position change*
- *LCR - logical connector replacement*
- *LOD - logical operator deletion*
- *LOR - logical operator replacement*
- *ROR - relational operator replacement*
- *SCD - super calling deletion*
- *SCI - super calling insert*
- *SIR - slice index remove*

Operadores experimentais (não utilizados por padrão):

- *CDI - classmethod decorator insertion*
- *OIL - one iteration loop*
- *RIL - reverse iteration loop*
- *SDI - staticmethod decorator insertion*
- *SDL - statement deletion*
- *SVD - self variable deletion*
- *ZIL - zero iteration loop*

A ferramenta *mutpy* foi desenvolvida como parte de uma dissertação de mestrado e a última atividade no repositório oficial da ferramenta data de novembro de 2019.

6. Avaliação experimental

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, de tal forma que o conjunto de casos de teste final estivesse adequado tanto para critérios de teste estrutural como teste de mutação.

Para isso foi realizado um estudo de adequação para preparar o ambiente de execução dos programas do *benchmark* com as ferramentas de teste investigadas. O repositório do projeto possui uma estrutura individual para cada programa, contendo um arquivo *json* que contém os dados de entrada e saída esperada para os casos de teste de cada programa e um *script* de teste responsável por ler os dados e realizar as asserções de cada programa.

A execução dos dados de teste populados nos arquivos *json* é feita por meio das diretivas `@pytest.mark.parametrize`, por meio das quais é possível definir uma serie de argumentos e *fixtures* na função ou classe de teste, sendo possível a parametrização dos casos de teste.

A definição da estrategia utilizando *script* de teste e dados de teste separados permite modularizar cada vez mais os nossos testes, deixando o projeto cada vez mais claro, com uma sintaxe simplificada dos casos de teste, e um outro arquivo trabalhando na leitura desses casos, possibilitando futuramente a inserção de novos casos de teste com um menor esforço.

Para cada uma das ferramentas investigadas foi projetado um *script* de teste que controla a execução e a coleta dos resultados.

Nas subseções a seguir apresentamos os resultados específicos para cada uma das ferramentas analisadas e as atividades realizadas para obtenção de um conjunto de casos de teste adequado para os critérios de teste investigados nesse projeto.

6.1. *pytest*

Os resultados obtidos com a execução do *script* de teste para coleta das informações contidas nos conjuntos de casos de teste podem ser observados na Tabela 2.

Com base nos resultados obtidos no relatório de cobertura 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.

Nos casos de teste dos programas `knapsack.py` e `levenshtein.py`, os casos de teste abaixo apresentavam problemas que inviabilizavam a execução dos programas:

`knapsack.py`:

- [[6404180, [[382745, 825594], [799601, 1677009], [909247, 1676628], [729069, 1523970], [467902, 943972], [44328, 97426], [34610, 69666], [698150, 1296457], [823460, 1679693], [903959, 1902996], [853665, 1844992], [551830, 1049289], [610856, 1252836], [670702, 1319836], [488960, 953277], [951111, 2067538], [323046, 675367], [446298, 853655], [931161, 1826027], [31385, 65731], [496951, 901489], [264724, 577243], [224916, 466257], [169684, 369261]]], 13549094]

`levenshtein.py`:

- [["amanaplanacanalpanama", "docnoteidissentafastneverpreventsafatnessidieton-cod"], 42]

Tabela 2. Cobertura gerada pelo *pytest* utilizando critério de cobertura de linhas

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`	8	0	2	0	100%
`is_valid_parenthesization`	9	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%
`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`	13	0	8	0	100%
`sieve`	6	0	5	0	100%
`sqrt`	5	0	2	0	100%
`subsequences`	7	0	6	0	100%
`to_base`	9	0	2	0	100%
`topological_ordering`	7	0	8	0	100%
`wrap`	8	0	2	0	100%
Total	**373**	**3**	**193**	**1**	**99%**

Tal problema está relacionado com as características dos programas, que utilizam funções recursivas. Para os casos de teste acima, o tempo de execução dos programas era demasiadamente alto, comprometendo a rotina de execução dos demais testes.

Na versão atual do *benchmark* os casos de teste foram ajustados utilizando uma função que determine um tempo máximo de execução para cada caso de teste executado. Dessa forma, ao esgotar o tempo de execução estimado para o teste, ele passa a ser automaticamente encerrado.

Para o programa `sqrt.py`, 2 casos de testes não apresentaram o resultaram esperado definido no conjunto de casos de teste:

- [[27, 0.01], 5.196164639727311]
- [[33, 0.05], 5.744627526262464]

Foi identificado que pelo fato de o programa realizar operações utilizando números ponto flutuante, a depender do sistema operacional da máquina que está executando o teste, o resultado produzido pelo programa pode diferenciar por algumas casas decimais, fazendo com que o teste não seja verificado da forma correta.

Para solucionar esse problema foi utilizada uma funcionalidade disponível na biblioteca do *unittest* (assertAlmostEqual), que é capaz de comparar números ponto flutuante especificando o número de casas decimais de precisão para o teste. Dessa forma, com a utilização de tal abordagem foi possível atingir a cobertura total do programa.

Para os programas `depth_first_search` e `shortest_path_length` o relatório inicial não apresentou uma cobertura de 100% devido ao conjunto de testes propostos inicialmente não exercitar as linhas de código especificadas na Figuras 4 e 5 .

Coverage for **python_programs/shortest_path_length.py** : 97%

30 statements | 29 run | 1 missing | 0 excluded

```
1  from heapq import *
2
3  def shortest_path_length(length_by_edge, startnode, goalnode):
4      unvisited_nodes = [] # FibHeap containing (node, distance) pairs
5      heappush(unvisited_nodes, (0, startnode))
6      visited_nodes = set()
7
8      while len(unvisited_nodes) > 0:
9          distance, node = heappop(unvisited_nodes)
10         if node is goalnode:
11             return distance
12
13         visited_nodes.add(node)
14
15         for nextnode in node.successors:
16             if nextnode in visited_nodes:
17                 continue
18
19             insert_or_update(unvisited_nodes,
20                 (min(
21                     get(unvisited_nodes, nextnode) or float('inf'),
22                     distance + length_by_edge[node, nextnode]
23                 ),
24                 nextnode)
25             )
26
27     return float('inf')
28
```

Figura 4. Trechos de código não cobertos pelos casos de teste - Parte 1

Resumidamente, os trechos de código referentes à linha 17 do programa `shortest_path_length` e a linha 7 do programa `depth_first_search` não foram excitadas por nenhum caso teste, pois, nenhum caso de teste utilizou o conceito de que dado um nó ele já está contido no conjunto de nós visitados. Podemos concluir que para que alcance 100% de cobertura, é necessário a criação de alguns casos de teste que explorem essa característica especifica dos programas.

Apesar de apresentar uma cobertura de 79%, observa-se pelos resultados que o `node` teve todas as arestas cobertas pelos casos de teste propostos. Essa é uma deficiência da ferramenta *pytest-cov*, que possui dificuldades em coletar corretamente os rastros de execução em métodos de acesso a atributos de classes, conforme descrito em uma *issue*

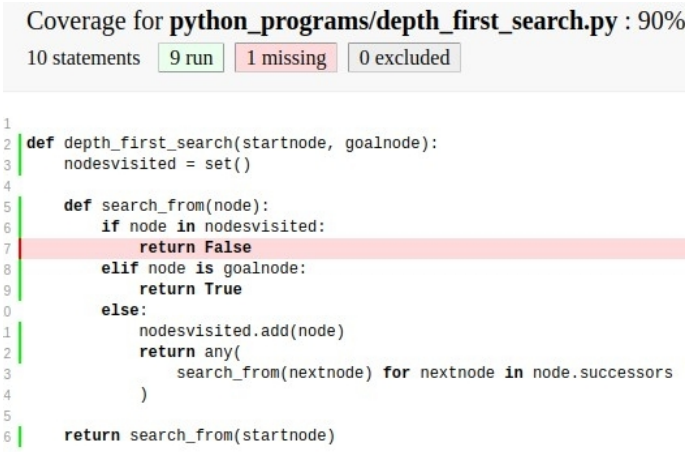


Figura 5. Trechos de código não cobertos pelos casos de teste - Parte 2

#277 [2] no código-fonte do repositório da ferramenta.

Para o programa `next_permutation`, nós observamos que trata-se de um caminho inatingível [Barbosa et al. 2016], ou seja, não há um caso de teste capaz de gerar a execução produzida pela aresta especificada no relatório de cobertura.

Apesar do *pytest* ser uma ferramenta estável e bastante popular no meio da comunidade *Python*, durante o desenvolvimento da atividade que compreende esse relatório foram encontradas algumas dificuldades com relação à utilização da ferramenta. Tais dificuldades abrem espaço para possíveis melhorias futuras. Essa subseção descreve as principais dificuldades encontradas e como elas foram superadas.

6.1.1. 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 necessita uma versões da linguagem de programação *Python* 3.5 ou superior [Holger Krekel 2020].

Durante o desenvolvimento das atividades que compreendem esse relatório foi observado que alguns projeto utilizados fazem uso de características especificas da linguagem *Python* 3, como por exemplo trechos de código no programa powerset.py:

- first, *rest = arr

  Uma conversão equivalente do código para a versão *Python* 2 seria:

- first = arr[0]
  rest = arr[1:]

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.

[2] `https://github.com/pytest-dev/pytest-cov/issues/277`

Devido a essa limitação, os casos de teste para o programa powerset.py apresentavam comportamento inesperado, uma vez que para a sua correta execução fazia-se necessária a utilização do *Python* 3. Tendo em vista esse problema, optou-se por migrar todo o projeto para execução em um ambiente virtual *Python* isolados, que possibilitaria controlar com precisão a versão das bibliotecas e da linguagem de programação.

Decidimos utilizar o ambiente virtual (*virtualenv*) pois serve se solução para problemas que envolvem dependências, versões e, indiretamente, problemas de permissões. Para o nosso cenário, problema com as versões. O *virtualenv* cria um ambiente que possui seus próprios diretórios de instalação, que não compartilham bibliotecas com o sistema operacional ou outros ambientes virtuais, além de ser um mecanismo para separar diferentes ambientes *Python* para repositórios distintos.

Após a instalação do *virtualenv* criamos o ambiente Pyhton independente, com o nome venv onde foi instalado uma versão do *setuptools* e do gerenciador de pacotes (*pip*).

Foi instalada a versão *Python* 3.6.9, e adicionamos um arquivo de texto chamado `requirements.txt`, contendo todas as bibliotecas necessárias para a execução do projeto, de forma a possibilitar de uma forma simples a reprodução dos dados apresentados nesse relatório.

Com essa solução foi possível resolver o problema relacionado à versão do *Python* que o *pytest* utilizava para executar os testes.

6.1.2. 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 fazer uma rápida pesquisa na internet [Google 2020] é possível observar que existem diversas dúvidas sobre como organizar os 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.

Ao utilizar o *pytest*, nos deparamos com o seguinte cenário: como separarmos as partes do projeto em diretórios específicos. Por exemplo, "diretório dos programas", "diretório com os casos de testes", "diretório com o conjunto de testes no formato *json*", entre outros.

Para solucionar esse problema, no nosso projetos, optamos por utilizar uma variável global que identifica o diretório central do projeto e por meio dela identificar os demais diretórios, como o diretório onde se encontra os casos de testes, o diretório onde estão os programas, entre outros necessários.

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 3.2). Assim como na subseção anterior, o objetivo dessa etapa consiste em

avaliar o estado inicial do conjunto de casos de teste fornecido primariamente pelo *benchmark* 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 5.3, 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*.

A Tabela 3 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 estrutural todas-as-arestas, discutido na subseção 6.1.

Tabela 3. Resultados para a ferramenta de teste de mutação *cosmic-ray*

Programa	Total	Mortos	Sobreviventes	Taxa de Sobr. (%)
bitcount	18	18	0	0,00%
breadth_first_search	11	10	1	9,00%
bucketsort	28	28	0	0,00%
depth_first_search	10	9	1	10,00%
detect_cycle	16	15	1	6,25%
find_first_in_sorted	97	90	7	7,22%
find_in_sorted	79	73	6	7,59%
flatten	3	3	0	0,00%
gcd	19	18	1	5,26%
get_factors	78	67	11	14,10%
hanoi	66	65	1	1,52%
is_valid_parenthesization	31	28	3	9,68%
kheapsort	2	2	0	0,00%
knapsack	101	99	2	1,98%
kth	53	52	1	1,89%
lcs_length	52	51	1	1,92%
levenshtein	53	53	0	0,00%
lis	83	77	6	7,23%
longest_common_subsequence	36	36	0	0,00%
max_sublist_sum	18	17	1	5,56%
mergesort	54	49	5	9,26%
minimum_spanning_tree	10	9	1	10,00%
next_palindrome	135	121	14	10,37%
next_permutation	101	99	2	1,98%
pascal	91	88	3	3,30%
possible_change	44	44	0	0,00%
powerset	23	23	0	0,00%
quicksort	44	43	1	2,27%
reverse_linked_list	1	1	0	0,00%
rpn_eval	46	46	0	0,00%
shortest_path_length	64	59	5	7,81%
shortest_path_lengths	16	16	0	0,00%
shortest_paths	28	19	9	32,14%
shunting_yard	26	24	2	7,69%
sieve	35	34	1	2,86%
sqrt	80	80	0	0,00%
subsequences	70	68	2	2,86%
to_base	52	51	1	1,92%
topological_ordering	5	5	0	0,00%
wrap	23	22	1	4,35%
Total	**1802**	**1712**	**90**	4,65%

Por meio da realização dessa atividade foi possível observar as características e limitações da ferramenta *cosmic-ray*, que são discutidas em maiores detalhes a seguir:

A ferramenta possui um mecanismo que evita, automaticamente, a execução de mutantes tidos como problemáticos. De acordo com a documentação oficial [3], a ferramenta examina o módulo do programa em teste para identificar todas as mutações possíveis, mas nem sempre decide por executar todas elas. Por exemplo, por identificar que algumas dessas mutações resultarão em mutantes equivalentes e acaba marcando essas mutações como "*skipped*".

Na Tabela 3 é possível observar que, em geral, ouve uma baixa taxa de mutantes sobreviventes, de tal forma que os casos de teste gerados na subseção anterior foram suficientes para mata a grande maioria dos mutantes gerados.

Dentre os aspectos negativos observados na utilização da ferramenta está o mais crítico 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*[4] 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.

6.2.2. *mutmut*

Conforme descrito na subseção 5.4, 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*.

A Tabela 4 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 estrutural todas-as-arestas, discutido na subseção 6.1.

Por possuir um menor conjunto de operadores de mutação e, consequentemente, gerar um menor número de mutantes, os casos de teste projetados durante a etapa da seção 2, foram suficientes para matar todos os mutantes gerados para essa ferramenta.

Um ponto especifico 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*. Por conta disso, essa ferramenta não foi capaz de gerar mutantes para o programa `flatten`.

[3] `https://cosmic-ray.readthedocs.io/en/latest/filters.html`
[4] `https://github.com/sixty-north/cosmic-ray/issues/486`

Tabela 4. Cobertura gerada pelo *mutmut*

Programas	Total	Mortos	Timeout	Suspeitos	Sobreviventes	Skip
bitcount	9/9	7	2	0	0	0
breadth_first_search	7/7	7	0	0	0	0
bucketsort	8/8	8	0	0	0	0
depth_first_search	5/5	5	0	0	0	0
detect_cycle	10/10	9	1	0	0	0
find_first_in_sorted	23/23	21	2	0	0	0
find_in_sorted	13/13	13	0	0	0	0
flatten	-	-	-	-	-	-
gcd	3/3	3	0	0	0	0
get_factors	12/12	8	4	0	0	0
hanoi	15/15	15	0	0	0	0
is_valid_parenthesization	15/15	15	0	0	0	0
kheapsort	1/1	1	0	0	0	0
knapsack	19/19	17	2	0	0	0
kth	12/12	12	0	0	0	0
lcs_length	10/10	10	0	0	0	0
levenshtein	17/17	17	0	0	0	0
lis	21/21	20	1	0	0	0
longest_common_subsequence	13/13	13	0	0	0	0
max_sublist_sum	8/8	8	0	0	0	0
mergesort	23/23	20	3	0	0	0
minimum_spanning_tree	5/5	5	0	0	0	0
next_palindrome	39/39	32	7	0	0	0
next_permutation	21/21	20	1	0	0	0
node	6/6	5	1	0	0	0
pascal	21/21	21	0	0	0	0
possible_change	12/12	10	2	0	0	0
powerset	5/5	5	0	0	0	0
quicksort	11/11	11	0	0	0	0
reverse_linked_list	5/5	4	1	0	0	0
rpn_eval	15/15	15	0	0	0	0
shortest_path_length	19/19	18	1	0	0	0
shortest_path_lengths	6/6	6	0	0	0	0
shortest_paths	8/8	7	1	0	0	0
shunting_yard	15/15	13	2	0	0	0
sieve	7/7	7	0	0	0	0
sqrt	12/12	10	2	0	0	0
subsequences	11/11	11	0	0	0	0
to_base	12/12	11	1	0	0	0
topological_ordering	4/4	4	0	0	0	0
wrap	8/8	5	3	0	0	0
Total	**486**	**449**	**37**	**0**	**0**	**0**

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 causa a ocorrência de *loops* infinitos.

É importante ainda apontar com relação às características dessa ferramenta, é 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.

6.2.3. *mutpy*

Conforme descrito na subseção 5.5, 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*.

Tabela 5. Cobertura gerada pelo *mutpy*

Programas	Todos	Mortos	Sobreviventes	Incompetentes	Timeout	Cobertura
bitcount	7	5	0	0	2	100%
breadth_first_search	3	3	0	0	0	100%
bucketsort	9	3	0	6	0	100%
depth_first_search	3	3	0	0	0	100%
detect_cycle	4	4	0	0	0	100%
find_first_in_sorted	25	22	0	1	2	100%
find_in_sorted	19	18	0	1	0	100%
flatten	1	0	0	1	0	100%
gcd	4	3	0	0	1	100%
get_factors	15	10	4	1	0	100%
hanoi	15	13	0	2	0	100%
is_valid_parenthesization	15	15	0	0	0	100%
kheapsort	3	3	0	0	0	100%
knapsack	17	16	1	0	0	100%
kth	13	13	0	0	0	100%
lcs_length	9	9	0	0	0	100%
levenshtein	25	25	0	0	0	100%
lis	18	17	1	0	0	100%
longest_common_subsequence	19	18	0	1	0	100%
max_sublist_sum	4	4	0	0	0	100%
mergesort	27	25	1	1	0	100%
minimum_spanning_tree	2	2	0	0	0	100%
next_palindrome	24	23	0	1	0	62.6%
next_permutation	27	26	1	0	0	100%
node	-	-	-	-	-	-
pascal	19	19	0	0	0	100%
possible_change	13	12	1	0	0	100%
powerset	3	1	0	2	0	100%
quicksort	13	10	1	2	0	100%
reverse_linked_list	1	1	0	0	0	100%
rpn_eval	16	16	0	0	0	100%
shortest_path_length	20	15	1	4	0	100%
shortest_path_lengths	4	4	0	0	0	100%
shortest_paths	6	5	1	0	0	100%
shunting_yard	21	21	0	0	0	100%
sieve	8	8	0	0	0	100%
sqrt	16	13	0	0	3	100%
subsequences	11	10	0	1	0	100%
to_base	10	5	0	3	2	100%
topological_ordering	4	4	0	0	0	100%
wrap	10	7	1	0	2	100%
Total	**483**	**431**	**13**	**27**	**12**	**99,06%**

A Tabela 5 apresenta os resultados 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 na subseção 2, também foram acrescidos novos casos de teste para matar os mutantes que haviam inicialmente sobrevivido.

A coluna Incompetentes refere-se a mutantes que ao serem executados geram algum tipo de exceção *(raised TypeError)*, semelhante à ferramenta *mutmut*, a coluna *timeout* refere-se a mutantes que são capazes de executar os casos de teste, mas se mantém em um período de execução muito alto, potencializando a presença de *loops* infinitos e a coluna sobreviventes refere-se ao mutantes que após análise individual não puderam ser mortos, sendo identificados, portanto, como equivalentes.

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 no *Apêndice* deste documento.

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

Conforme apontado na seção 3, um dos fatores, apontados como limitantes, para adoção pratica do teste de mutação em contextos industriais está relacionado ao seu alto custo computacional, 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.

A Tabela 6 apresenta o tempo de execução gasto em segundos para cada uma das ferramentas. É possível observar que, conforme citado na subseção 6.2.1, o custo computacional para execução dessa ferramenta 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 execução 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.

Tabela 6. Tempo de execução (em segundos) dos programas do *benchmark* utilizando as ferramentas de teste de mutação

Programas	cosmic	mutmut	mutpy
bitcount	56,09	80,2	11,45
breadth first search	27,54	0,61	1,02
bucketsort	33,51	0,63	1,82
depth first search	27,58	0,66	0,96
detect cycle	27,25	40,45	1,06
find in sorted	136,59	0,59	14,05
first in sorted	142,9	80,26	3,39
flatten	54,58	0,63	0,76
gcd	81,53	2,21	6,05
get factors	161,16	16,74	3,55
hanoi	162,81	0,6	4,56
is valid parenthesization	90,93	0,59	2,91
kheapsort	56,25	0,66	0,97
knapsack	196,05	5,66	3,07
kth	128,22	0,6	2,65
lcs length	128,53	0,61	2,26
levenshtein	169,96	0,61	25,41
lis	207,1	4,85	3,63
longest common subsequence	128,66	0,57	5,78
max sublist sum	88,36	0,59	1,27
mergesort	171,76	84,4	11,4
minimum spanning tree	75,14	0,59	0,92
next palindrome	239,88	28,61	3,45
next permutation	256,6	4,87	5
node	9,86	4,88	0
pascal	287,1	0,74	3,18
possible change	470,78	8,7	6,74
powerset	33,13	0,76	0,93
quicksort	19,68	0,63	2,83
reverse linked list	10,57	5,15	0,64
rpn eval	27	0,62	2,95
shortest path lengths	32,63	0,58	3,7
shortest path length	16,82	4,9	1,17
shortest paths	19,27	6,08	1,37
shunting yard	17,56	8,62	4,05
sieve	12,11	0,56	1,81
sqrt	112,26	80,2	18,07
subsequences	135,47	0,59	3,12
to base	123,98	40,47	12,02
topological ordering	71,74	0,6	1,09
wrap	114,64	84,91	11,76
Total	**4363,58**	**605,78**	**192,82**

6.2.5. Considerações finais

Para o uso da técnica de teste estrutural, é necessário o conhecimento da estrutura interna do programa, para que assim, a partir do grafo de fluxo de controle (GFC), pode se aplicar critérios de testes, criando-os para a obtenção dos resultados. Podem existir elementos dentro do GFC que não são executáveis, funcionalidades que nenhum teste consegue exercitá-la, com isso gerando um relatório sobre o programa não tão concreto, mas como visto, seu custo não inviabiliza sua aplicação pois mostra um bom resultado.

Para o uso da técnica de teste de mutação, além do conhecimento da estrutura

interna do programa para identificar defeitos típicos que podem ocorrer, é necessário investigar aspectos que vão além de métricas relacionadas a cobertura de código, sendo necessário observar as possíveis mutações, surgidas a partir dos operadores de mutação, modeladas no programa, as quais variam de acordo com a linguagem e a ferramenta utilizada, conforme foi possível observar nos resultados deste trabalho, com isso os testes podem revelar diferentes tipos de defeitos. Apesar do seu custo computacional ser mais elevado em relação ao teste estrutural, o teste de mutação mostrou-se um critério altamente eficaz em revelar defeitos para que assim possa melhorar ainda mais a qualidade de conjuntos de casos de teste e refinar ainda mais os programas buscando aprimorar aspectos relacionados a qualidade e confiabilidade.

7. Conclusão

Esse relatório apresentou os resultados do trabalho de iniciação cientifica, desenvolvido pela aluna durante o período de 09/04/2019 a 08/10/2020. Os resultados da avaliação experimental desenvolvida pela aluna 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.

Como principal contribuição desse trabalho, foi disponibilizado, de forma publica, um repositório[5] 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.

Dentre as principais capacidades e habilidades desenvolvidas pela aluna durante o período dessa iniciação cientifica, destaca-se a compreensão de algoritmos escrito na linguagem de programação *Python*, o aprimoramento das competências sobre teste de software, pois durante a graduação são poucas as matérias que dão a oportunidade de trabalhar com tais assuntos (programação em *Python* e teste de software), a busca por soluções de erros que surgiam referente ao ambiente proposto (trabalhar com algoritmos em *Python* juntamente com as ferramentas de teste), e saber lidar com situações em outra língua, pois diversas vezes a solução do problema não se encontra acessível na língua materna.

Assim, tais capacidades e habilidades puderam contribuir para a sua formação da seguinte forma: saber situar-se no contexto inserido, no caso, sobre teste de software para linguagem de programação *Python*, possibilitando com isso elevar a compreensão de técnicas e práticas de teste de software propostas nas literaturas (estrutural e de mutação) e sobre a linguagem de programação em si, visto que é uma linguagem que tende a crescer no mercado devido a sua compatibilidade em diversas aplicações (desenvolvimento web, inteligência artificial, big data e estruturas de teste integrados).

Dessa forma, saber identificar a maneira que novas ferramentas e tecnologias trabalham e como utilizá-las para atingir determinados resultados pode contribuir para o desenvolvimento de uma análise mais crítica e criteriosa na área. Por fim, vários desafios e obstáculos foram enfrentados durante o desenvolvimento das etapas da iniciação científica e, com isso, foram aprimoradas habilidades de pensamento crítico, pesquisa e resoluções de problemas, características fundamentais para a formação da aluna.

[5] `https://github.com/RenataBrito/quickbugs_report/`

Referências

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.

Codecademy education company (2020). Codecademy python courses & tutorials. Available from: https://www.codecademy.com/catalog/language/python.

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

Google (2020). how to do imports from another folder in python? Available from: https://bit.ly/2x7T7XV.

Guido van Rossum (2019). Python language. Available from: https://gvanrossum.github.io/.

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

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.

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

Offutt, A. J. and Untch, R. H. (2001). Mutation 2000: Uniting the orthogonal. In *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, pages 34–44, San Jose, California.

Pressman, R. and Maxim, B. (2016). *Software Engineering: A Practitioner's Approach (Eighth Edition, English Abridgement)*. Ji xie gong ye chu ban she.

StackOverflow (2019). Developer survey results 2019. Available from: https://insights.stackoverflow.com/survey/2019.

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 (2020). Tiobe index for march 2020. Available from: https://www.tiobe.com/tiobe-index/.

Apêndice

Find first in sorted

Foram adicionados os casos de teste descritos abaixo para matar, respectivamente, os mutantes (a) e (b) da Figura 6.

- [[[314, 962, 456, 159], 314], 0]
- [[[1], 1], 0]

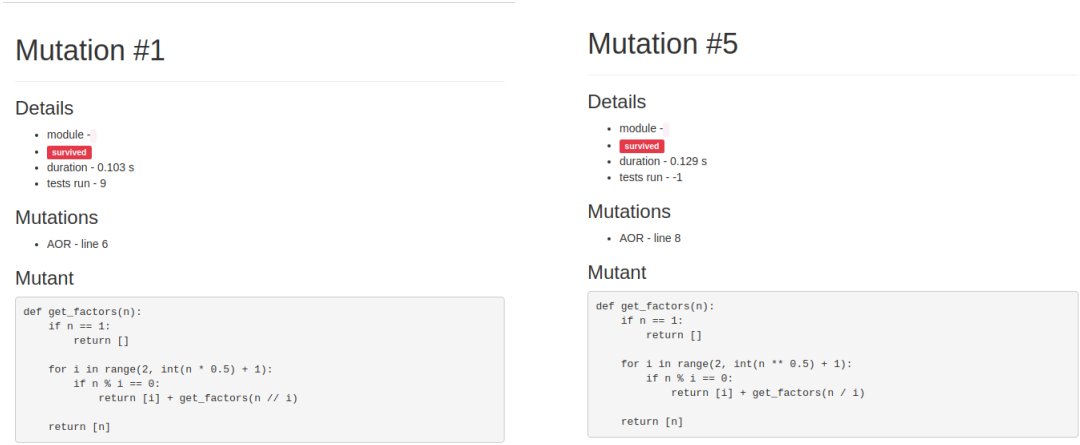
Figura 6. Novos mutantes mortos para o programa *Find first in sorted*.

(a) CRP, `lo = 1` (b) CRP, `mid == 1`

Get factors

Foram identificados 4 mutantes equivalentes descritos nas Figuras 7 e 8.

Figura 7. Mutantes equivalentes para o programa *Get factors*.



(a) AOR, `n * 0.5` (b) AOR, `n/1`

Figura 8. Mutantes equivalentes para o programa *Get factors* #2.

(a) CRP, `int(n ** 1.5)`

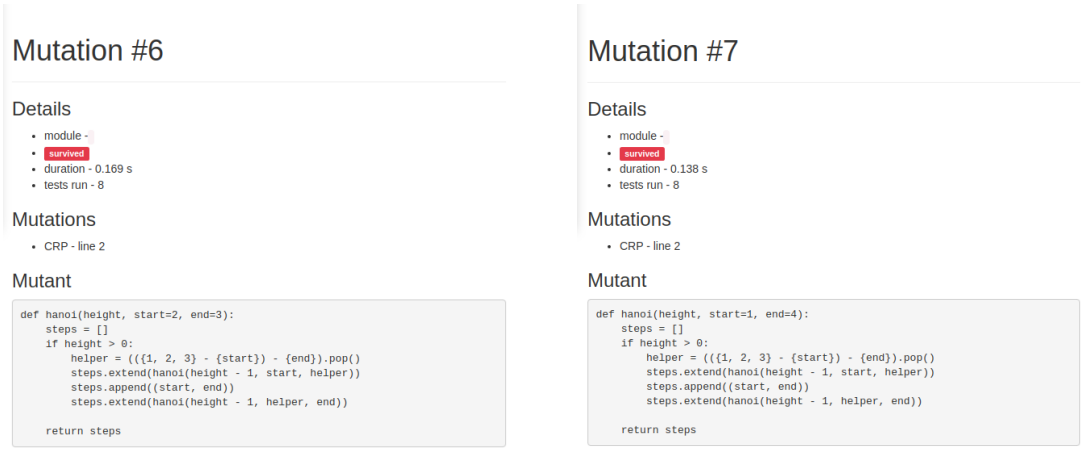
(b) CRP, `int(n ** 0.5) + 2)`

Hanoi

Foram adicionados novos casos de teste para matar os mutantes da Figura 9.

- [[3], [[1, 3], [1, 2], [3, 2], [1, 3], [2, 1], [2, 3], [1, 3]]]

Figura 9. Novos mutantes mortos para o programa *Hanoi*.



(a) CRP, `start = 2 end = 3`

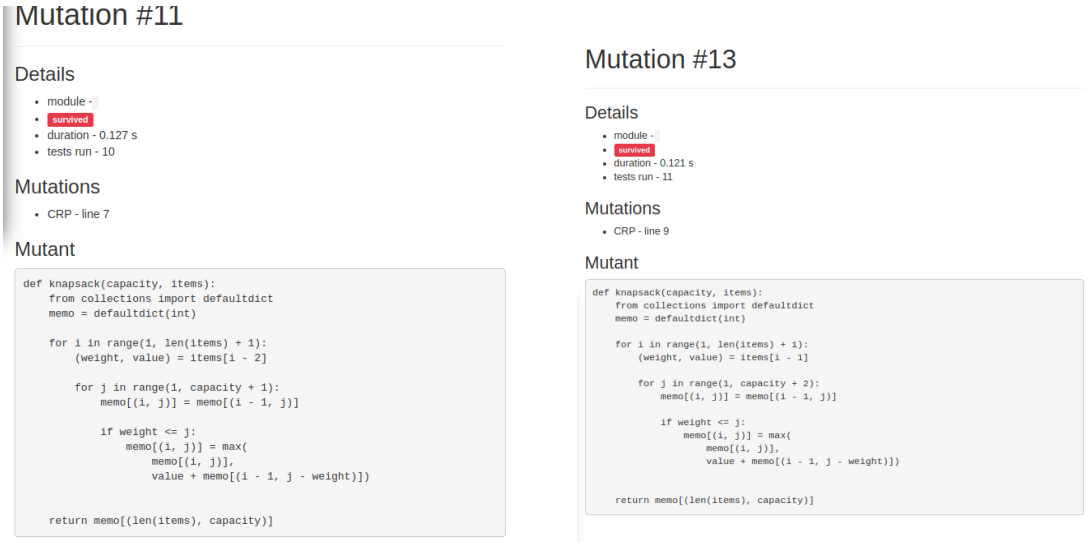
(b) CRP, `start = 1 end = 4`

Knapsack

Foi adicionado um novo caso de teste para matar 1 mutante e foi identificado 1 mutante equivalente, representados, respectivamente, em (a) e (b) na Figura 10.

- [[43, [[0, 100], [25, 28], [18, 49]]], 149]

Figura 10. Novo mutante morto e mutante equivalente para o programa *Knapsack*.



(a) CRP, `capacity + 2`

(b) CRP, `items(i - 2)`

Levenshtein

Foi adicionado um novo caso de teste para matar o mutante da Figura 11.

- [[’test’, ’estt’], 2]

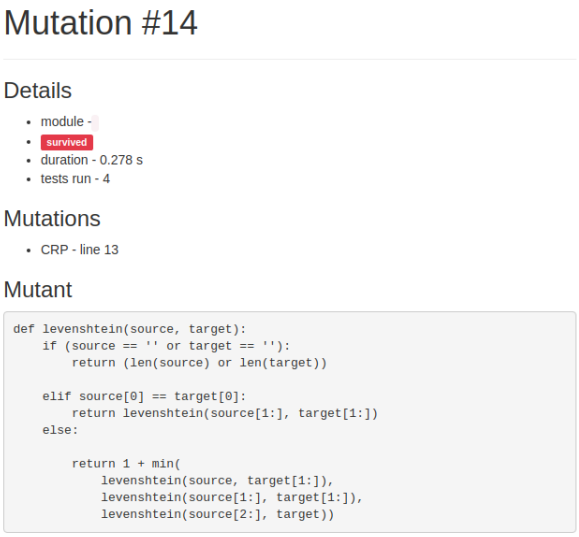


Figura 11. Novo mutante morto para o programa *Levenshtein* | `(CRP, source[2:])`.

Lis

Foi identificado 1 mutante equivalente, representado na Figura 12.

Figura 12. Novo mutante equivalente para o programa *Lis* | `(ROR, val <= arr)`.

Mergesort

Foi identificado 1 mutante equivalente, representado na Figura 13.

Figura 13. Novo mutante equivalente para o programa *Mergesort* | `(ROR, left[i] < right[j])`.

Next permutation

Foi identificado 1 mutante equivalente, representado na Figura 14.

Figura 14. ROR, perm[i] <= perm[i + 1]

Possible change

Foi adicionado um novo caso de teste para matar 1 mutante e foi identificado 1 mutante equivalente, representados, respectivamente, em (a) e (b) na Figura 15.

- [[[1, 0.25], 0.5], 1]

Figura 15. Novo mutante morto e mutante equivalente para o programa *Possible change*.

(a) CRP, `total < 1`

(b) ROR, `total <=`

Quicksort

Foi identificado 1 mutante equivalente, representado na Figura 16.

Figura 16. Novo mutante equivalente para o programa *Quicksort* | `(SIR, for x in arr[:] if x < pivot.`

Rpn eval

Foi adicionado um novo caso de teste para matar o mutante da Figura 17.

- [[[0.5, 3.0, 2.0, "/", "+"]], 2.0]

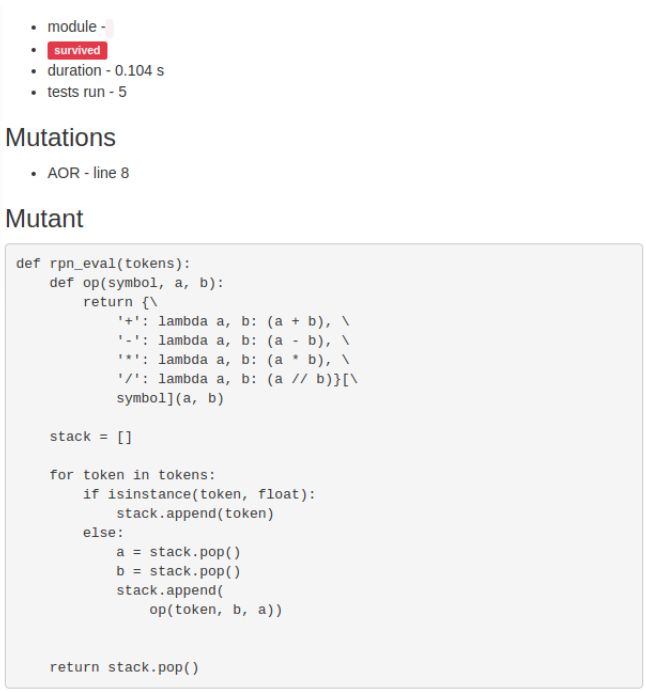
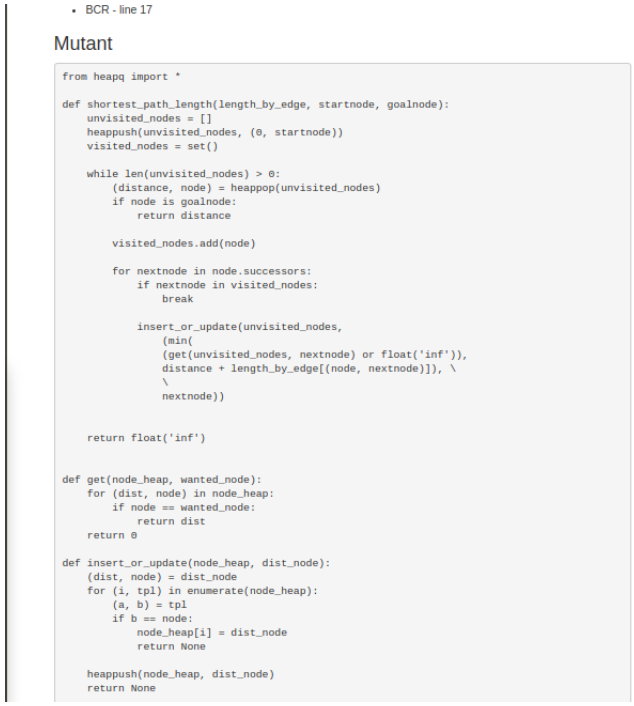


Figura 17. Mutante morto, *Rpn eval* | `AOR, '/': lambda a, b: a // b.`

Shortest path length

Foi identificado 1 mutante equivalente, representado na Figura 18.



- BCR - line 17

Mutant

```
from heapq import *

def shortest_path_length(length_by_edge, startnode, goalnode):
    unvisited_nodes = []
    heappush(unvisited_nodes, (0, startnode))
    visited_nodes = set()

    while len(unvisited_nodes) > 0:
        (distance, node) = heappop(unvisited_nodes)
        if node is goalnode:
            return distance

        visited_nodes.add(node)

        for nextnode in node.successors:
            if nextnode in visited_nodes:
                break

            insert_or_update(unvisited_nodes,
                (min(
                (get(unvisited_nodes, nextnode) or float('inf')),
                distance + length_by_edge[(node, nextnode)]), \
                \
                nextnode))


    return float('inf')


def get(node_heap, wanted_node):
    for (dist, node) in node_heap:
        if node == wanted_node:
            return dist
    return 0

def insert_or_update(node_heap, dist_node):
    (dist, node) = dist_node
    for (i, tpl) in enumerate(node_heap):
        (a, b) = tpl
        if b == node:
            node_heap[i] = dist_node
            return None

    heappush(node_heap, dist_node)
    return None
```

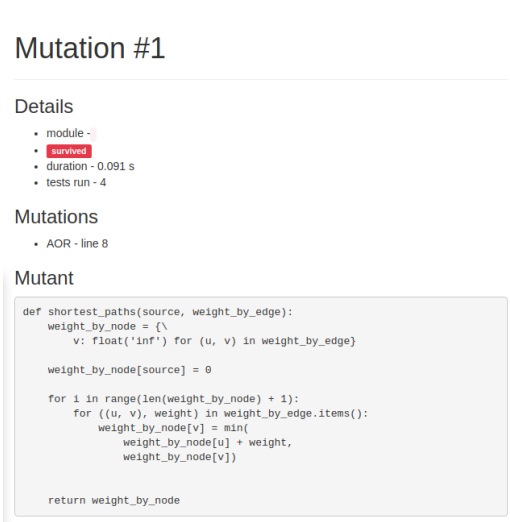
Figura 18. Novo mutante equivalente (*Shortest path length*) | `(BCR, if nextnode in visited_nodes: break`.

Shortest paths

Foi identificado 1 mutante equivalente e adicionado um novo caso de teste para matar 1 mutante. Representados, respectivamente, em (a) e (b) na Figura 19.

- def test_Graph_with_on_node(): Output = 'B':1, 'A':0 graph4 = ('A', 'B'): 1, assert shortest_paths('A', graph4) == Output

Figura 19. Novo mutante morto e mutante equivalente para o programa *Possible change*.



Mutation #1

Details

- module -
- survived
- duration - 0.091 s
- tests run - 4

Mutations

- AOR - line 8

Mutant

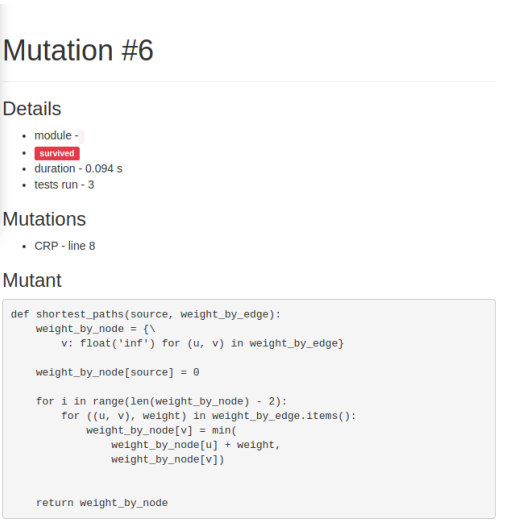
```
def shortest_paths(source, weight_by_edge):
    weight_by_node = {\
        v: float('inf') for (u, v) in weight_by_edge}

    weight_by_node[source] = 0

    for i in range(len(weight_by_node) + 1):
        for ((u, v), weight) in weight_by_edge.items():
            weight_by_node[v] = min(
                weight_by_node[u] + weight,
                weight_by_node[v])


    return weight_by_node
```

(a) AOR, `(weight_by_node) + 1`



Mutation #6

Details

- module -
- survived
- duration - 0.094 s
- tests run - 3

Mutations

- CRP - line 8

Mutant

```
def shortest_paths(source, weight_by_edge):
    weight_by_node = {\
        v: float('inf') for (u, v) in weight_by_edge}

    weight_by_node[source] = 0

    for i in range(len(weight_by_node) - 2):
        for ((u, v), weight) in weight_by_edge.items():
            weight_by_node[v] = min(
                weight_by_node[u] + weight,
                weight_by_node[v])


    return weight_by_node
```

(b) CRP, `(weight_by_node) - 2`

Shunting yard

Foi adicionado um novo caso de teste para matar o mutante da Figura 20.

- [[[10, "*", 2, "/", 2,"*",4]], [10, 2, "*", 2, "/", 4, "*"]]

Figura 20. CRP, '*': 3

Sqrt

Adicionados dois novos casos de teste para matar os mutantes das Figura 21.

- [[2.5,0.01], 1.5817307692307692]
- [[16,9.0], 5.0]

Figura 21. Novo mutante mortos para o programa *Sqrt*.

(a) AOR, `x//2`

(b) ROR, `>= epsilon`

Wrap

Foi identificado 1 mutante equivalente e adicionado um novo caso de teste para matar 1 mutante. Representados, respectivamente, em (a) e (b) na Figura 22.

- [["a a a", 2],["a", "a", "a"]]

Figura 22. Novo mutante equivalente e mutante morto para o programa *Wrap*.

(a) AOR, `(weight_by_node) + 1`

(b) CRP, `(weight_by_node) - 2`