Search-based Test Data Generation for Mutation Testing: a tool for Python programs

Matheus M. Ferreira[1], **Lincoln M. Costa**[2], **Francisco Carlos M. Souza**[1]

[1]Departament of Software Engineering, Federal University of Technology - Paraná, Dois Vizinhos-PR, Brazil

[2]Computer Systems Engineering Program, Federal University of Rio de Janeiro, Rio de Janeiro-RJ, Brazil

matheusf.2016@alunos.utfpr.edu.br,costa@cos.ufrj.br,

franciscosouza@utfpr.edu.br

Abstract. *Test data generation for mutation testing consists of identifying a set of inputs that maximizes the number of mutants killed. Mutation Testing is an excellent test criterion for detecting faults and measuring the effectiveness of test data sets. However, it is not widely used in practice due to the cost and complexity to perform some activities as generating test data. Although test suites can be produced and selected manually by a tester this practice is susceptible to errors and tools are needed to facilitate it. Several tools have been developed to automate mutation testing, but, only a few address the test data generation. The present paper proposes an automated test data generation tool based on weak mutation for Python programming language using the Hill Climbing algorithm. For evaluation, we performed an empirical study concerning the effectiveness and cost computational of the tool in a database composed of 348 mutants and we compare it with random generation. Overall, the experiment achieved an average mutation score of 86% for our proposed tool and random testing 64% on average.*

1. Introduction

Mutation testing is the most common type of software fault-based testing and this is based on producing hypothetical faulty programs by creating variants of the program under test (PUT). This criterion was proposed by [DeMillo and Offutt 1991] for detecting faults and measuring the effectiveness of tests. The process works from faults that are injected into the program under test to produce faulty program versions called mutants. In general, to perform software testing using any technique is required to create a set of test cases to execute the artifact under test, be it code, interfaces, or requirements. A test case is composed of a pair of, *i)* test data which are inputs to execute the program under test and *ii)* expected output. Thus, test data generation is an activity focused on finding valid input according to specific test criteria.

Mutation testing is widely considered as effective and powerful. However, this criterion also is known as extremely costly, mainly due to three factors: i) the high number of mutants produced, ii) the number of test data necessary to identify the mutants, and iii) the difficulty of finding the equivalent mutants which are mutants with the same behavior regarding the original program. The number of test data to detect a fault into a mutant is

related to the strategy applied for generating them since if the test data set is generated manually can be labor-intensive and susceptible to errors task [Papadakis and Malevris 2010].

The main objective of the mutation testing is to generate a test data set that can reveal the faults in the mutant programs so that they fail, that is, to distinguish the outputs of the mutant programs from the original ones. The test data generation is an activity that has been a growing interest in the community due to being hard and it not having reached a high level of automation. Thus, an approach able to completely automate this activity is an important step to reduce mutation testing costs and increase more reliability in the tests. For these reasons intelligent approaches to solving this problem have been arising amongst them the utilizing of search-based algorithms.

Mutation testing is considered to be a powerful test criterion, due to its effectiveness in revealing faults. Nevertheless, it is a very expensive technique, for this reason, alternatives have emerged to reduce its costs which were named, weak mutation, firm mutation, and strong mutation that refers to the traditional mutation testing. In this case, mutants killed in this context is said as mutants strongly killed. In weak mutation, mutants are killed if immediately after the execution of the mutated statements there is a state difference between the original and mutant programs, and they are called weakly killed mutants. In Firm mutation, the aim is to verify if there is a difference between the states of the original and mutant programs at a later point after the execution of the mutation point, the more differences, the more likely the mutant has to be killed by a test die.

In this context, the paper presents an automated approach for generating test data based on the weak mutation to strongly kill mutants using a Hill Climbing algorithm for programs written in the Python language. In this study, we utilized an objective function derived from a study proposed by Weneger et al. [Wegener et al. 2001] and applied for the same context in [Souza et al. 2016] called Reach distance and Mutation distance. The difference from the previous works is that it aims at generating test data to kill mutants in Python, the frameworks in the literature only automatize the process of mutants production and tests execution that shows there is still the problem in this area.

2. Mutation testing for Python

Mutation testing aims to verify whether the program under test is not present defects and also assess the quality of the test suite. Assessing of the test sets, faulty versions of the program original are produced, which simulate the mistakes made by programmers, these versions are defined as mutants. Hence, the goal is to execute test data that causes the mutants to behave differently from the original program. A test data that identifies a fault makes the mutant be considered dead, otherwise, it is said to be alive [DeMillo and Offutt 1991]. However, there are two possibilities if a mutant remains alive after executing each input of the test data set. The first one, the mutant is considered equivalent, i.e,. for all inputs, the mutant will produce the same output as the original program. The second possibility is that the test set is weak to kill the mutant, i.e., improvements are necessary to carry out the mutants identifications. To generate the mutants, it is necessary to consider the features of a programming language, since the process for generating mutants is performed though applying mutation operators. The mutant operators are usually based on typical errors that occur during the software development and they determine the type

of syntactic change that must be made to generate mutants, such as command mutations, operator mutations, variable mutations, and constant mutations.

Performing mutation testing depends largely on the existing tools to automate its process since applying this technique manually is impracticable. Despite producing mutants and generating test sets can be performed non-automatic by a tester, this practice is complex and time-consuming. For Python programming language, according to Derezinska and Halas [Derezinska and Halas 2014] the first mutation tool was created in 2002 based on a Java mutation tool Jester and currently has emerged better tools, but not sufficiently automated to perform all mutation testing activities as can be seen in Table 1.

Currently, we consider four main tools to test python programs based on mutation testing and they are defined as Cosmic Ray, MutMut, MutPy, PyMuTester. Cosmic Ray is a mutation testing tool for Python 3 and it has been successfully used on a variety of projects ranging such as assemblers to oil exploration software. MutMut is a mutation testing system also for Python 3 it focuses mainly on ease of use and supports to all test runners. MutPy is a tool for Python programs from 3.3+ version, it uses the standard unittest module, generates YAML/HTML reports, and has colorful output, also supports high order mutations (HOM) and code coverage analysis. PyMuTester is a tool to execute mutation testing, its main purpose is to assist faults in if-condition negation and loop skipping

Table 1 shows information about whether the tools can produce mutants (second column), they generate test data sets automatically (third column), execute the test sets against the mutants (fourth column), if the tools have some features to analyze equivalent mutants by the tester (fifth column), the number of mutant operators available (sixth column) and the year of the first and the last version (seventh and eighth column). As we can see, the features analyzed in the tools, most of them can produce mutants and execute the tests against mutants, however, none of them generate test data sets, i.e., the inputs must be produced manually to execute the tests.

Table 1. Python Mutation Testing Tools

Tools	Generate Mutants	Generate Test sets	Execute Test	Marking of Eq. Mutants	Mutant Operators	First Version	Current Version
Cosmic Ray	Yes	No	Yes	Yes	14	2017	2020
MutMut	Yes	No	Yes	No	12	2016	2020
MutPy	Yes	No	Yes	No	17	2011	2019
PyMuTester	Yes	No	Yes	No	2	2011	2017

3. Related work

Several publications have appeared in recent years documenting different techniques for test data generation in mutation testing, these studies address the use of search-based algorithms such as Hill Climbing [Souza et al. 2016], particle swarm optimization (PSO) [Jatana et al. 2016] and, genetic algorithm (GA) [Rani et al. 2019]. There are also a few studies specifically about mutation testing for Python [Derezinska and Halas 2015] and [Derezinska and Halas 2014]. However, as far we know no research addresses automatic test data generation for this context.

Papadakis and Malevris [Papadakis and Malevris 2010] proposed an approach that generates test data to kill weak and strong mutants using Dynamic Symbolic Exe-

cution (DSE) for Java programs. The approach transforms the original program into a meta-program, containing all the weak-mutant-killing constraints, and then the test data to cover all the branches the meta-program are generated through DSE. Thus, the DSE produces test data to strongly kill the mutants automatically based on weak-mutant-killing constraints. The work concludes that a high level of automation of the generation of test cases for killing the mutants can be achieved.

According to Souza et al. [Souza et al. 2016], a Hill Climbing algorithm was used to generate the test data for weak and strong mutants for programs written in C. In this study, an objective function involving three parts was used to find the best result. The first one, with the help of Branch Coverage Testing, is guided to the test data to reach the mutation point. The second function is used to assist in the generation of data test that can infect the mutation point, and the third one makes this infection to impact the flow of the program in such a way that the fault is propagated to the output. More recently, Rani et al. [Rani et al. 2019] proposed a genetic algorithm to generate the test cases automatically for mutation testing. This research employed a selective mutation technique to minimize the mutation testing cost i.e, to create and execute fewer mutants instead of all the traditional mutation operators. An experiment was performed out from a set of mutants in Java language.

4. Approach Description

Test data generation is a fundamental activity to improve the automatizing of mutation testing. These automatizing consists mainly of mutants production, test data generation, programs, and mutants execution using the test sets and test oracle. Excepts for test data generation, the mentioned activities are already automated in several tools. A test data is considered as an effective one if it kills one or more mutants, by producing different outputs from the original program and mutants, thus checking whether the test set can identify the changes injected into the source code. In general, to distinguish the behavior between original program P and a mutant M, a test data t should satisfy three conditions known RIP model, Reachability, Infection, and Propagation [DeMillo and Offutt 1991]:

1. Reachability: t must be able to reach the original statement S and mutated statement S'. If S and S' are not reached by t, t is not guaranteed to kill M.
2. Infection: t must be able to cause different internal states on P and M immediately after executing S and S'. Thus, for t to kill M, it is necessary that S and S' are reached and the state after its execution must be different.
3. Propagation: t must be able to cause the final state of the M to be different from P, i.e., the infected state must propagate to some point in P at which it can be observed. Thus, the different state caused by satisfying the necessity condition must be propagated through the program's execution to produce a different output.

In this context, the proposed approach attempts to automate the mutation-based test data generation for Python programs. This project aims to create a library that utilizes search-based algorithms to produce test data sets and can be employed in different mutation testing tools. As we can see in Figure 1, the process is composed of the following steps: *(1)* searching and generating of test data; *(2)* executing the test suite against mutants; *(3)* analyzing the outputs by the test oracle, removing the killed mutants and

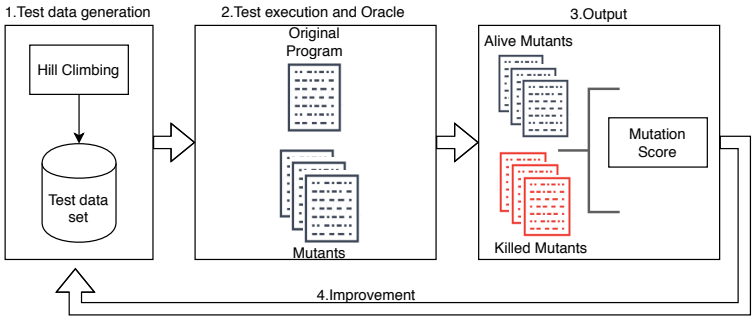

Figure 1. Scheme of the proposed tool

computing of mutation score; e *(4)* return and improving the generated test data from Hill Climbing. This process is repeated until reaching a predefined score threshold.

Hill Climbing is a search algorithm that combines a general search method, i.e., generate and test solutions through objective functions to evaluate the states produced by the method. This algorithm aims to identify the best path to be followed in the search and then, it returns the optimal or a satisfactory result for a given problem. Therefore, the algorithm consists of selecting an initial solution randomly, evaluating it, and improving it step by step, from the investigation of the neighborhood of the current solution [Russell and Norvig 2009].

An important element to achieve optimal results in search-based algorithms is how to assess the candidate solutions, this element is described as objective functions. Objective functions consist of mathematical functions that guide the search to find the best solutions and they are created using specific information on the problem. In this paper, we utilize an objective function based on RIP conditions, more specifically in Reachability and Infection and it is composed into two parts: reach distance and mutation distance, as used in [Souza et al. 2016]. The first part of the function is regarding Reachability (RD) and it guides the search towards the mutation point. The second one called MD is a reference to infection condition and it aims to infect the program state at the mutation point, i.e., making a difference in the execution between the two program versions.

The approach consists of generating test data for Python programs from a Hill Climbing algorithm based on weak mutation, i.e., considering reachability and infection condition to kill strongly mutants. The objective function is composed of Reach distance and Mutation distance which represents the weak mutation. Reach distance is a function that combines two metrics and it has been widely used in previous studies [Korel 1990], [McMinn and Holcombe 2006], and [Papadakis et al. 2010]. This function is formed by metrics used in structural testing described as approach level and branch distance. Approach level measures the distance that a test data needs to cover the targeted statement using the number of the target mutant's control dependent nodes that were not executed by the test data. For nodes in which the execution flow was diverted, the distance for a branch to be taken as true, and it is performed from the values of the variables or constants in the condition statements, this metric is called Branch distance.

The second part of our objective function defined as mutation distance was proposed by Papadakis et. al [Papadakis and Malevris 2013] and it is a generalization of the study introduced by Bottaci [Bottaci 2001] in which a fitness function for a genetic

algorithm in mutation testing was presented. Mutation distance computes how close test data are to expose a difference between the original and mutated statement according to branch distance.

For the tool to handle mutations, we deal with the codes on an Abstract Syntax Tree (AST) and reach branches using branch coverage. Assuming a mutant as a branch, the aim is to generate test data to traverse a tree, in other words, a test data that achieves coverage of the mutant branch means that the reachability condition has been fulfilled. Whether this condition has not been fulfilled the process continues until to reach the mutation. A phase of test data improvements is started utilizing the distance necessary for an input reaching the target branch and this is calculated with branch distance and approach level metrics as presented in [Wegener et al. 2001]. After the reachability condition is achieved, we verify using the same test data generated if the infection condition has been fulfilled. For achieving infection state a test data should lead to a different state on a mutant program in the mutation point and original statement. Finally, if the infection is achieved for a selected mutant, the test data are executed in all mutants to verify how many have been strongly killed. The process continues to improve the test data generate until killed all mutants.

The approach above mentioned was developed into a tool that is accessed using a command terminal as well as the reports produced as illustrated in Figure 2. The main features of the tool is about the following items: i) –function, ii) –method, iii) –int-min and –int-max as presented in Table 2. The tool was developed using as a base a Python implementation of automated test data generation for branch testing [1]. From this implementation, we can deal with mutants as branches to apply weak mutation, thus we developed an extension to execute the original programs and mutant programs to verify if they have been killed from a given test data. It is worth mentioning that our tool has no mutants production, for the experiment we use an external tool.

Table 2. Flags arguments to execute the proposed tool

Flag argument	Function
–function <target function name>	define the function to be tested
–method <Hill Climbing>	define search algorithm
–int-min	minimum value of initial parameters for the technique
–int-max	maximum value of initial parameters for the technique

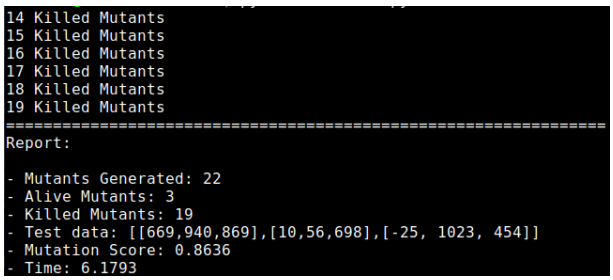

Figure 2. Report from the proposed tool

[1] https://pypi.org/project/covgen/

5. Experimental Study

We conducted experiments to analyze and evaluate the proposed tool for test data generation on a set of Python programs. In this study, the guidelines recommended by Wholin et al. [Wohlin et al. 2012] were used. The experiment was performed through a laptop with Intel Core i7 2.4GHz CPU, 8GB memory in Ubuntu 19.10 operating system.

5.1. Experiment Definition

We used the Goal-Question-Metric (GQM) model [Basili and Weiss 1986] to set out the objectives of the experiment that can be summarized as follows: *"Analyse **proposed tool** for the purpose of **evaluation** with respect to **the mutation-based test data generation** from the point of view of **experimenters** in the context of **the Python programs**"*.

For achieving the goal, we investigate the following Research Question (RQ): **How effective is the proposed tool for test data generation to kill mutants in Python programs?** To answer this RQ, the effectiveness of the tool was measured using the mutation-based test data generation for eight Python programs. We also performed this experiment 30 times computing the average mutation score and the number of test data.

5.2. Procedure of Experiment

To answer the RQ, we carried out the experiment, as follows: *(i)* generating test data to kill the mutants using the proposed approach; and *(ii)* computing mutation score. These experiments were performed in three steps: (1) we chose eight Python programs P = (p_1, p_2,..., p_8) of different size as experimental subjects. Table 3 presents name and LOC of the programs; (2) we generate mutants using mutpy tool[2]; (3) we generated the test data to kill the mutants using the proposed tool; and (4) the total of mutation score is computed.

Table 3. Python programs used in the experiment

Programs	LOC
boolop	25
calender	94
changerMoney	43
coordinates	30
grades	18
orelse	17
triangle	37
typeTriangle	17

5.3. Results

In this section, we answer the RQ presented in Section 5 from the analysis of results concerning the effectiveness of the proposed approach. For this, we compare the proposed tool with a random generation. Table 4 shows the number of mutants (second column), test data generation using proposed tool (T) e (R) random generation (fourth and fifth column), the number of alive mutants by (T) e (R) in sixth column and the number of killed mutants by (T) e (R) in last one.

Analyzing the last column is possible to notice that the performance of the proposed tool was very significant since it killed more than random testing in all programs.

[2]https://pypi.org/project/MutPy/

Table 4. Comparison between the number of killed mutants by the proposed tool (T) and random generation (R)

Programs	Mutants	Equivalentes	Test Data		Alive		Killed	
			T	R	T	R	T	R
boolop	37	3	8	7	4	13	30	21
calendar	87	7	6	11	4	21	76	59
changerMoney	64	8	7	10	9	24	47	32
cordinates	52	4	4	8	12	21	36	27
grades	33	5	4	6	8	11	20	17
orelse	22	3	3	4	0	6	19	13
triangle	20	2	3	5	3	6	15	12
typeTriangle	33	2	5	8	3	7	28	24
Total	348	34	40	59	43	109	271	205
Average	35	3.5	4.5	7.5	4	12	29	22.5

We compared the proposed tool with random testing by evaluating the number of the test data generated capable of killing the largest number of mutants (RQ). The program with the most test data was *calendar* and the program with minimum test data was *orelse*. As expected, the bigger program required more test data than the smaller ones. As presented in Table 4, test data generation using the proposed tool is more efficient than a random generation, since it generates 19 fewer test data with high quality able to kill 77.81% of all mutants. Thus, it was clear that the random generation was unable to kill the whole set of mutants since most of the test data produced are not sufficient.

Figure 3 shows the mutation score achieved by the proposed tool and random generation against the subject programs. As the results show, the proposed tool achieved a mutation score of 22% more than the random generation. In all programs, independent of size, it was possible to observe that the proposed tool obtained better mutation scores than the random generation. Also, we compute the time to generate test data sets and execute it against the set of mutants for each program. Table 5 presents the average time obtained by our tool, the first column indicates the name of programs, the second column reports the average mutation score achieved per program, and the third column shows the average time required for the whole execution.

Table 5. Time and mutation score

Programs	Mutation Score	Time
boolop	0.88	7.0164
calendar	0.95	8.0102
changerMoney	0.84	8.184
cordinates	0.75	6.5821
grades	0.71	7.9134
orelse	1	6.1793
triangle	0.83	5.3258
typeTriangle	0.9	6.7134
Average	0.86	6.8649

We can notice that the test data set generated based on reachability and infection condition is adequate to strongly kill mutants. However, even our benchmark be composed of simple programs, for some mutants are necessary more efforts to find adequate test data, this is due to the difficulty of satisfying more complex requirements or difficulty for a test data to propagate the wrong state to the program's output. Thus, we notice that

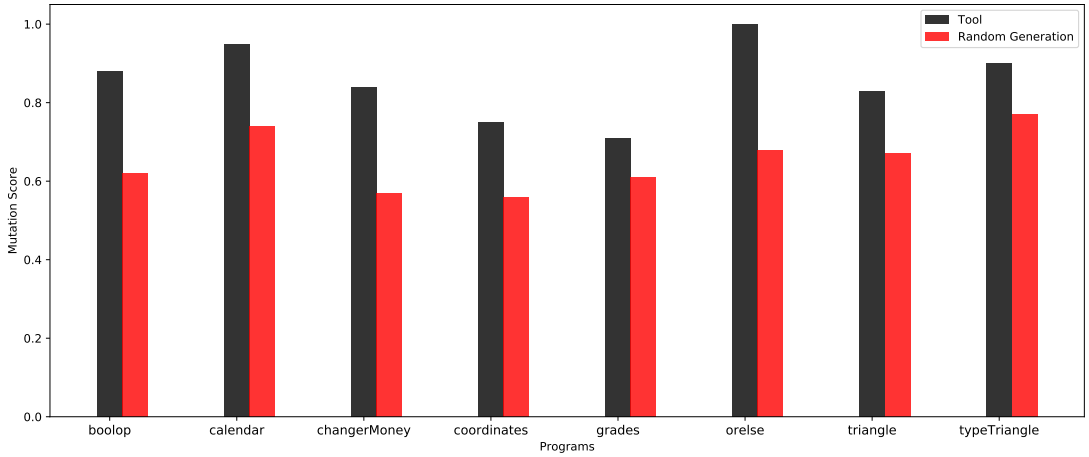

Figure 3. Comparison of MS achieved between the tool and random generation

it is important to add in our objective function the propagation condition.

6. Conclusion

Test data generation is an important activity in software testing, the goal is generating a large number of test data to fulfill testing criteria. Although this activity is known to be performed manually by a Tester, it demands a lot of effort and automation is necessary. For several years a great effort has been devoted to the study of techniques and methods to address this issue, but only a few tools are considered sufficiently adequate to be applied in industrial environments. Generating test data based on mutation for python programs is still a gap, the existing tools focus on mutants generation, engines to execute the test data against mutants and test oracles. An automated approach can lead to the many opportunities to produce quality test data sets and ensuring the cost reduction for mutation testing.

The paper proposes an attempt to automate the test data generation activity through a tool. Our tool is based on approaches presented in previous studies as [Papadakis and Malevris 2013], [Fraser and Arcuri 2015] and [Souza et al. 2016] which use search-based algorithms to generate adequate test data employing the RIP conditions. For guiding the test generation we employed an objective function based on weak mutation, i.e., it is composed of Reachability and Infection conditions to kill strongly mutants. From our preliminary experimental, the results indicate that for toy Python programs the tool was able to kill 86% on average of all mutants and 22% more than random generation. Future work comprises in *i)* conducting additional experiments using real programs, *ii)* increase the impact condition on the objective function and *iii)* finally, we will attempt to provide a tool for Python programs that can be employed in different mutation testing tools which no generates test data automatically.

References

Basili, V. and Weiss, D. (1986). A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(6):728–738.

Bottaci, L. (2001). A genetic algorithm fitness function for mutation testing. In *Proceedings of the First International Workshop on Software Engineering using Metaheuristic Innovative Algorithms*, pages 3–7, Toronto, Ontario, Canada. IEEE Computer Society.

DeMillo, R. A. and Offutt, A. J. (1991). Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17:900–910.

Derezinska, A. and Halas, K. (2014). Experimental evaluation of mutation testing approaches to python programs. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 156–164.

Derezinska, A. and Halas, K. (2015). Improving mutation testing process of python programs. In *CSOC*.

Fraser, G. and Arcuri, A. (2015). Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812.

Jatana, N., Suri, B., Misra, S., Kumar, P., and Choudhury, A. R. (2016). Particle swarm based evolution and generation of test data using mutation testing. In *Computational Science and Its Applications – ICCSA 2016*, pages 585–594. Springer International Publishing.

Korel, B. (1990). Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879.

McMinn, P. and Holcombe, M. (2006). Evolutionary testing using an extended chaining approach. *Evolutionary Computation*, 14(1):41–64.

Papadakis, M. and Malevris, N. (2010). Automatic mutation test case generation via dynamic symbolic execution. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 121–130.

Papadakis, M. and Malevris, N. (2013). Searching and generating test inputs for mutation testing. *Springer Plus*, 2(1):1–12.

Papadakis, M., Malevris, N., and Kallia, M. (2010). Towards automating the generation of mutation tests. In *Proceedings of the 5th Workshop on Automation of Software Test (AST)*, pages 111–118, Cape Town, South Africa. ACM.

Rani, S., Dhawan, H., Nagpal, G., and Suri, B. (2019). Implementing time-bounded automatic test data generation approach based on search-based mutation testing. In *Progress in Advanced Computing and Intelligent Engineering*, pages 113–122. Springer Singapore.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*.

Souza, F. C. M., Papadakis, M., Le Traon, Y., and Delamaro, M. E. (2016). Strong mutation-based test data generation using hill climbing. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, pages 45–54. ACM.

Wegener, J., Baresel, A., and Harmen, S. (2001). Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854.

Wohlin, C., Runeson, P., Host, M., Ohlsson, M. C., Regnell, B., and Wesslen, A. (2012). *Experimentation in Software Engineering: An Introduction*. Springer-Verlag Berlin Heidelberg, 1st. edition.