Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Evaluating Automatic Program Repair Approaches for Python

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von:	Kai Werk
geboren am:	16.08.1995
geboren in:	Berlin
Gutachter/innen:	Prof. Dr. Lars Grunske
	Dr. Thomas Vogel
eingereicht am:	....
verteidigt am:	....

Abstract.
Objective Automatic Program Repair has gained significant attention as a means to reduce cost and effort involved in debugging software. This thesis evaluates various APR approaches for Python, including GENPROG, CARDUMEN, KALI and MUTREPAIR.
Method To compare the effectiveness of these approaches, we designed an experimental study based on the student assignment benchmark, investigating the repair rate, runtime, and differences in the search space. Given the inherent randomness of some of the approaches, we applied statistical hypothesis testing to ensure the robustness of our findings. Furthermore, we evaluated the impact of genetic programming on APR to understand its potential benefits and limitations in this context.
Results PYGENPROG finds the most patches among all approaches. However, each approach is able to find patches for certain subjects that no other approach can repair, justifying their individual relevance. The impact of genetic programming on APR is not as strong as initially expected and requires further optimization to improve its effectiveness.

Contents

1 Introduction

Software bugs can have severe consequences. A recent example is the CrowdStrike incident in July 2024, where a faulty update to the Falcon software caused widespread system failures. This led to massive disruptions in sectors such as finance, healthcare, and aviation, with estimated financial damages of 4-6 billion dollars [15]. To prevent such incidents, developers invest significant time in debugging and fixing errors, a process that is both time-consuming and costly [27].

To improve debugging efficiency, various automated techniques have been developed. Some focus on explaining bugs [10, 11, 21], while others aim to localize [1, 19, 20, 53, 59, 60] them. However, the actual process of fixing these issues remains largely manual. This is where Automatic Program Repair (APR) comes into play, enabling the automated correction of software bugs without human intervention.

There are many different APR approaches. Some use evolutionary search algorithms to generate patches, while others rely on predefined patterns to expand the search space and target specific types of bugs. However, no single approach is equally effective for all defects. Each technique has unique strengths and weaknesses depending on the nature of the bug. This makes combining multiple approaches, known as APR collections, a promising direction for research.

One such collection is FIXKIT, a framework for evaluating various APR methods for Python. Most previous research has focused on Java and C, while Python, one of the most widely used programming languages, has received comparatively less attention. However, Python is extremely versatile, with applications in web development, machine learning and data science. This makes APR equally important for this language. This work aims to implement and evaluate established APR approaches specifically for Python.

Our contributions As part of this thesis, established APR approaches were implemented in Python, including GENPROG, CARDUMEN, KALI, and MUTREPAIR. For evaluation, a new benchmark for APR was used, based on student programming assignments. A comprehensive evaluation was conducted to assess the strengths and weaknesses of each approach, with a particular focus on the impact of genetic programming on automatic program repair.

Paper Overview Section 2 provides the necessary background for the implementation of FIXKIT, covering topics such as genetic programming, fault localisation, minimisation and the different APR approaches. The implementation of CARDUMEN is described in detail in Section 3. The experimental design is outlined in Section 4, where the research questions, methodology, and experiments conducted for this study are presented. In the following section, section 5, we evaluate the approaches and discuss the results. Limitations and threats to the validity of the research are addressed in Section 6. Section 7 is a discussion of related work on associated problems, and finally, we conclude our paper in Section 8.

2 Background

This section provides an overview of the theoretical foundations of search-based automatic program repair and the approaches implemented in FIXKIT. Search-based automatic program repair can be divided into the following components: fault localization 2.2, genetic programming 2.1, and minimization 2.3. Each of these components is discussed in detail in its respective section. Subsequently, the various approaches from FIXKIT and their specific characteristics are described in detail. This includes, in particular, GENPROG 2.4.1, an evolutionary approach to automatic program repair that is considered one of the most prominent methods in this field of research. GENPROG has significantly contributed to establishing this research domain. Due to its pioneering role and widespread recognition, it is frequently used as a baseline for evaluating new approaches [6, 45, 47, 57, 64].
MUTREPAIR 2.4.2 and KALI 2.4.3 both adopt an exhaustive search approach instead of the evolutionary strategy employed by GENPROG. MUTREPAIR specializes in mutating suspicious if statements to alter control flow, whereas KALI focuses solely on removing statements. Rather than generating a comprehensive fix for the bug, KALI aims to expose weaknesses in test suites. CARDUMEN 2.4.4, on the other hand, uses an evolutionary approach combined with a large search space facilitated by template generation

2.1 Genetic Programming

In this chapter and the remaining thesis, a candidate is a representation of a program we try to automatically repair. A population is a collection of candidates. Genetic Programming is an evolutionary algorithm inspired by Darwinian's theory of biological evolution to automate the process of program generation [18]. There are many applications for genetic programming, including categorizing news stories, performing optical character recognition, and developing medical signal filters. [2]. The fundamental concept is to evolve computer programs through simulated evolutionary processes, where the goal is to find solutions that perform well on a given task. To find a solution, we need an initial candidate in the scope of automatic program repair that is the buggy program represented as an abstract syntax tree. The typical steps of genetic programming are

1. Deviate the candidates of the population
2. Evaluate the fitness of the candidates
3. Selection of the next generation

Deviate the candidates of the population There are several approaches to modifying the population in genetic programming. One such approach is mutation, where a candidate solution undergoes changes. In nature, small genetic deviations are beneficial for adapting to changing environments and gaining a competitive advantage over other species. In the context of Automatic Program Repair using genetic programming, mutation operators modify not genes, but code. The manner in which the code is mutated depends entirely on the mutation operator used. In GENPROG, for instance, mutation operators can insert, replace, or delete code fragments. While genetic mutations in nature occur largely by chance, in APR, the mutation process is guided and controlled through fault localization 2.2.

Another fundamental mechanism for altering genetic material is crossover, which in nature occurs through sexual reproduction. In genetic programming, crossover involves the exchange of parts between two genetic candidates. In the context of automatic program repair, this entails swapping portions of their respective syntax trees.

Evaluate the fitness of the candidates In nature, an individual's fitness is determined by its ability to survive - the longer an organism lives, the higher its fitness and the greater its chance of reproduction, allowing it to pass on its approved genes. In genetic programming, a fitness function is required to evaluate the quality of a candidate. The success of genetic programming is largely dependent on the design of this function. In Automatic Program Repair, the test suite serves as a crucial component for assessing a candidate's fitness. Simply put, the more test cases a candidate passes, the higher its fitness; conversely, failing more tests results in a lower fitness.

Selection of the next generation In nature, the fittest candidates are selected for reproduction. Similarly, genetic programming employs various selection mechanisms. One might wonder why we do not simply select the fittest candidates. The reason is to prevent the risk of converging too quickly to a local optimum, which could hinder further improvement. One commonly used selection method is the Roulette Wheel Selection. In this approach, the sum of all fitness values is computed, and candidates are selected based on weighted random selection. The probability of a candidate being chosen is determined by its fitness relative to the total fitness of the population. This method strikes a balance between exploration and exploiting fitter candidates, but it can also introduce a high degree of randomness [17]. Roulette Wheel Selection is the selection method used in FIXKIT and our experiments 4.3. Another notable selection method is Tournament Selection. In this approach, two candidates are randomly chosen, and the fitter one advances to the next generation. This method is computationally efficient, easy to implement, and highly parallelizable. Additionally, it ensures that weaker candidates still have a chance to survive and contribute to diversity within the population [2]. In practice, selection strategies often combine Elitism - where the best candidates automatically progress - with stochastic selection methods such as Roulette Wheel or Tournament Selection. This hybrid approach helps maintain strong solutions while promoting diversity, ultimately enhancing the effectiveness of genetic programming.

Challenges Despite the proven effectiveness of genetic programming, several challenges must be considered. One significant risk is converging to a local optimum rather than finding the global optimum. To mitigate this, various selection strategies are employed. Additionally, the search space in genetic programming is vast and nearly infinite, with countless possible mutations and combinations. Handling this complexity requires application-specific adaptations. For instance, CARDUMEN prioritizes particularly suitable templates and variable combinations for mutation, as discussed in Section 2.4.4. Another key challenge is bloat, where unnecessary mutations increase the size of candidates without being an essential step in the solution. Strategies for minimizing the final candidate while maintaining its ability to solve the problem are explored in Section 2.3.

2.2 Fault Localization

Identifying the optimal location for mutating a candidate is one of the primary challenges in automatic program repair. The underlying idea is that a statement consistently executed during undesired behaviour is likely the root cause of that behaviour and should be targeted for mutation. This process, known as fault localization, aims to pinpoint such faulty statements. Fault localization represents a crucial stage in the process of automatic program repair, as missing or inaccurate fault localization can result in incorrect or failed repairs.

The initial conceptualisations in this regard were put forth by Weiser, who proposed the technique of program slicing [59, 60]. A slice is a reduced part of the program which still behaves as the original program in a specific regard, i.e. failing a test case.

2.2.1 Spectra Fault Localization

The execution of a test suite can provide data beyond the identification of passed and failed tests. This data includes insights into the specific statements executed by a test case, which enables the localization of suspicious statements. This data is called program spectra and there are several types of spectra. These include block hit spectra, which are flags that are set when a particular code block is executed during a test case run, and block count spectra, which are counters that are incremented each time a code block is executed. A finer-grained scope can be used to examine individual statements. This method is known as spectrum based fault localization.

Tarantula is a popular approach in this domain. Tarantula [19, 20] assigns every statement a suspicious score from 0 to 1, with 1 being the most suspicious and 0 the least suspicious. The suspicious score is computed by the following equation:

$$suspiciousness(s) = \frac{\frac{failed(s)}{total\_failed}}{\frac{passed(s)}{total\_passed} + \frac{failed(s)}{total\_failed}} \tag{1}$$

The amount of failing test cases which execute statement s is described by $failed(s)$ and $total\_failed$ is the number of all failing test cases in the test suite. Contrary $passed(s)$ describes the amount of passing test cases which execute statement s and $total\_passed$ is the number of all passing test cases. The intuition behind this equation is that statements that are executed primarily by failing test cases are more likely to be faulty, statements that are executed primarily by passing test cases are less likely to be faulty. Contrary to other spectrum based techniques, Tarantula allows for some tolerance that the fault can be occasionally executed by a passed test. This tolerance shows positive impact on the fault localization.

Ochiai is a similarity coefficient originally used in molecular biology [43]. Abreu et al. [1] proposed that fault localization can be reduced to a similarity problem by representing the spectra as a hit matrix.

	Statement				
Test case	1	2	3	4	Error
1	×	×			×
2			×	×	
3	×		×		×
4				×	

Table 1: Similarity Matrix of Ochiai

In Table 1, each row represents a test case and indicates which statements were executed during its run. At the end of each row, an entry specifies whether the test case passed or failed. Viewing the table by columns, each column corresponds to a statement and shows in which runs that statement was executed. The last column exclusively shows the errors that occurred during execution. The goal is to identify the vector of a statement that most closely resembles the error vector, as it is hypothesized that this statement is responsible for the errors. To test their hypothesis, Abreu et al. borrowed Ochiai a similarity coefficient, originally used in molecular biology [43]. The coefficient is shown in Equation 2. The results were promising, as Ochiai proved to be more efficient than Tarantula. In fact, Ochiai is at least as effective as Tarantula and even surpasses it in specific cases.

$$suspiciousness(s) = \frac{failed(s)}{\sqrt{totalfailed \times (failed(s) + passed(s))}} \tag{2}$$

There are many more similarity coefficients, but Tarantula and Ochiai are the most prominent ones [62]. This is because they are simple to implement, yet effective. They are often used as a baseline to evaluate new coefficients.

There is no single approach that consistently outperforms all others across all scenarios. In other words, an optimal spectrum-based technique does not exist. This implies that the choice of technique must be tailored to each specific use case [65].

Spectrum based fault localisation is constrained by potential vulnerabilities in test cases. In the incident that a test case contains a bug, the intended functionality of the program will not be validated by the test cases, which may result in a wrong localization of the fault.

2.3 Minimisation

Automatic Program Repair often utilizes genetic programming, where random mutations and crossover operations are applied to genetic candidates, guided solely by a fitness function. While it is possible that the combination of these random mutations eventually leads to a correct program, the contributions of individual mutations can vary significantly. In some cases, every mutation may represent a small but crucial building block for the final solution. In other cases, the genetic algorithm might perform many irrelevant modifications before discovering a key mutation that ultimately results in a correct program.
In practice, the solution is often a mix: some modifications are essential to the program's correctness, while others are irrelevant. These irrelevant changes, often referred to as junk code, have no significant impact on the program's behaviour.
If this is the case, the patch needs to be minimized to include only the relevant changes. This not only enhances readability but also reduces unnecessary computations during every execution of the corrected program.

2.3.1 Delta Debugging

Delta Debugging [66] is a minimisation algorithm that applies the concept of binary search. The program is iteratively divided into smaller parts while ensuring that a desired property is preserved. In the original paper, test cases were minimized to help better understand the effects of specific faults. For our explanation of the *ddmin* algorithm, we apply this concept to mutations.

Ideal Scenario Assume an Automatic Program Repair approach using genetic programming finds a fix for the faulty program after four mutations. *Delta Debugging* then divides the mutations into subsets of two mutations each, as illustrated in Figure 2 First, the subset consisting of mutations 1 and 2 is tested, resulting in a failure of the test suite. Subsequently, mutations 3 and 4 are tested, and they pass the test suite. As a result, mutations 1 and 2 are excluded from the search space, as they have been identified as irrelevant to the repair. The successful subset, comprising mutations 3 and 4, is further divided into two subsets, each containing a single mutation. Mutation 3 is tested individually first but fails to satisfy the test suite. Afterwards, mutation 4 is tested and successfully passes the test suite. This process identifies a minimal mutation set, in this case consisting of a single mutation, which is sufficient to repair the faulty program. This represents an ideal scenario for delta debugging. However, what happens if the process deviates from this ideal scenario and a combination of mutations responsible for the fix is not immediately tested together within a subset?

Practical Example In order to consider a more comprehensive example, several terminologies need to be introduced. Let *mut* represent the original set of all mutations. Let mut' denote the set of mutations that remain in the current step. Let n represent the number of subsets into which the mutations are divided at each step. Initially, we set

		Mutations				
Step	Subset	1	2	3	4	test
1	Δ_1	1	2	.	.	×
2	Δ_2	.	.	3	4	✓
3	Δ_1	.	.	3	.	×
4	Δ_2	.	.	.	4	✓
Result		.	.	.	4	Done

Table 2: Ideal Scenario of Delta Debugging

$n = 2$.
In this practical example, four mutations representing a Patch for a buggy program are analysed, as illustrated in Figure 3. Initially, the algorithm tests the subsets (1,2) and (3,4). In this instance, the worst-case scenario occurs: both subsets fail. There are two possibilities to consider.

- **Increasing the subsets** increases the probability of finding a passing subset.
- **Decreasing the subsets** increases the progress if a passing subset is found, but the probability of finding one is lower.

Delta Debugging employs a combination of these two strategies. Initially, smaller subsets are tested by doubling the number of subsets, setting $n = 2n$. If one of these subsets passes, the process continues with that subset. If none of them pass, larger complements are created, defined as $\nabla_1 = mut' - \Delta_1$. In the example, the smaller subsets tested in steps 3, 4, 5, and 6 did not yield any successful results. However, the complement ∇_2 tested in step 8 is the first subset smaller than mut that satisfies the desired property: passing the test suite. At this point, the set of mutations is reduced to the complement, setting $mut' = \nabla_2$. Additionally, the granularity is decreased by one, such that $n = 3$.

What is the reason for decreasing the number of subsets n by exactly one? In each step, the subsets Δ have a fixed size. The complement ∇ is always mut' reduced by one Δ. This means reducing mut' to a specific complement while simultaneously decreasing n by one, reusing the original set from the previous step, is possible. These subsets have already been tested, allowing the algorithm to proceed directly with the new complements. The repetition of identical sets can be observed in the table 3 at step 3,5 and 6, as well as steps 9,10, and 11. The first new tests are introduced in steps 12 and 13 with the complements ∇_1 and ∇_2. In step 13 a new passing subset is identified. This subset is minimal, as all smaller subsets have already been tested and were all failing.

Based on the two examples in 2 and 3, all possible scenarios that can occur during minimisation with delta debugging were considered. Summarising, the rules that define the ddmin algorithm are as follows:

- **reduce to subset** if an Δ is found that satisfies the desired property, mut' is updated to Δ and reset n to two.

		Mutations					
Step	Subset	1	2	3	4	test	
1	Δ_1	1	2	.	.	×	$n = 2$
2	Δ_2	.	.	3	4	×	increase granularity $\rightarrow n = 2n$
3	Δ_1	1	.	.	.	×	n = 4
4	Δ_2	.	2	.	.	×	
5	Δ_3	.	.	3	.	×	
6	Δ_4	.	.	.	4	×	
7	∇_1	.	2	3	4	×	
8	∇_2	1	.	3	4	✓	reduce to subset $\rightarrow n = n - 1$
9	Δ_1	1	.	.	.	×	n = 3
10	Δ_2	.	.	3	.	×	
11	Δ_3	.	.	.	4	×	
12	∇_1	.	.	3	4	×	
13	∇_2	1	.	.	4	✓	found minimal passing subset
Result		1	.	.	4	Done	

Table 3: Practical Example of Delta Debugging

- **reduce to complement** if an ∇ is found that satisfies the desired property, mut' is updated to ∇ and n is decreased by one.
- **increase granularity** if neither a subset nor a complement satisfies the desired property, the granularity is doubled, i.e. $n = 2n$, provided that $n < |mut'|$.
- **otherwise** if none of the above cases applies, it implies that mut' is already minimal, and the process is complete.

2.3.2 Hierarchical Delta Debugging

Hierarichal Delta Debugging [44] (HDD) is a further development of Delta Debugging. This approach leverages the tree-structured nature of certain inputs, making it applicable exclusively to inputs that exhibit hierarchical structures, such as programming languages, HTML, or XML. The input is represented as a tree, which is feasible as long as the input can be described using a context-free grammar. The algorithm operates by traversing the tree from its root through multiple levels of nodes down to the tree's leaves. Each node is assigned to a specific level. At each level, the ddmin algorithm is applied to identify the minimal configuration. During this process, each node and its children are treated as a single unit that is not further differentiated at that level. Consequently, if a node is removed at a particular level, all its children are removed as well. This approach achieves significant minimization, particularly in the upper levels of the tree, as excluding a node also eliminates all its descendants. Another advantage of this algorithm is that every intermediate input remains syntactically correct. This effectively narrows the search space, focusing only on configurations that preserve a desired property, which inherently requires syntactic correctness.

2.3.3 HDDr

Hierarchical Delta Debugging recursively [23] (HDDr) introduces an optimization to HDD. In HDD, configurations often undergo testing that are derived from different, unrelated parts of the program. This issue arises due to the level-wise progression of the algorithm. In other words, when performing ddmin, HDD no longer uses the hierarchical structure of the input. To address this limitation, HDDr abandons the level-by-level traversal of the tree. Instead, it applies ddmin exclusively to sibling nodes. This approach offers two significant advantages: first, ddmin operates on smaller, more focused inputs; second, the inputs under consideration are logically related. As the name suggests, HDDr also incorporates recursion. Specifically, a node and all its children are minimized before moving to the next node.

2.4 Approaches Implemented in FIXKIT

2.4.1 GenProg

GENPROG employs genetic programming to find a repair for the faulty program [58]. Genetic programming is based on the principles of evolution. The objective is to identify the most fit candidate through the application of selection, crossover, and mutation [18]. GENPROG operates at the level of statements within a faulty program, utilizing the abstract syntax tree generated by that program. Therefore, the mutation operators employed for the mutation of a statement can replace a statement with another, remove a statement or insert a new statement [27]. The population is defined as all candidates in a generation. In each iteration of GENPROG, a random selection of candidates from the population is subjected to mutation. In addition, the mutation operator for each candidate is randomly selected []. GENPROG uses test cases and a fitness function to evaluate the fitness of a candidate. The fitness function is based on the number of failing tests that are fixed by the patch and the number of passing tests that are still passing with the patch. The fitness is defined as

$$\begin{aligned} \mathrm{F}_{\mathrm{GenProg}}(P) = & \; w_{PosT} \times |\{t \in PosT | P \text{ passes } t\}| \\ & + w_{NegT} \times |\{t \in NegT | P \text{ passes } t\}| \end{aligned} \tag{3}$$

where P is the patch, $PosT$ is the set of passing tests, $NegT$ is the set of failing tests, and w_{PosT} and w_{NegT} are the weights for the passing and failing tests, respectively. The candidates with the highest fitness are selected for continued evolution [58]. One disadvantage of genetic programming is that it can lead to irrelevant changes. To remove all unnecessary changes, delta debugging or structural differencing is applied to the remaining candidates at the end of the process [27]. In conclusion, the GENPROG algorithm comprises the following steps:

1. Fault Localization
2. Selection of Candidates

3. Selection of Mutation Operator for each Candidate
4. Evaluation of Population
5. Repeat 2.-4. until repair found
6. Removal of irrelevant changes

as shown in Figure 1. All the approaches follow the GENPROG main repair loop, but with key differences. These are highlighted in the corresponding section.

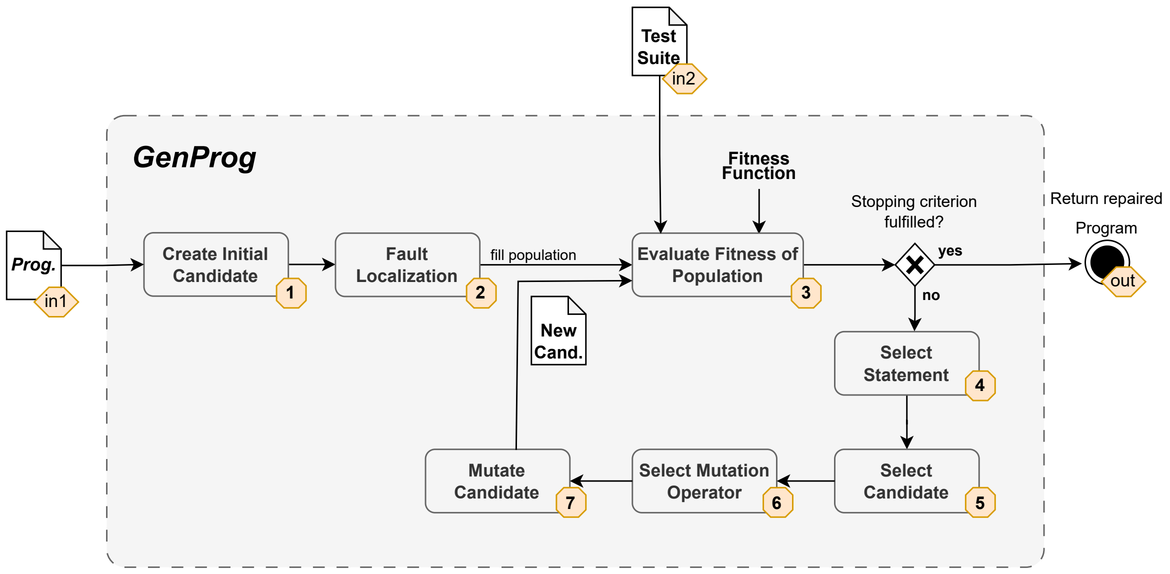


Figure 1: Overview of GENPROG algorithm

2.4.2 MutRepair

The MUTREPAIR algorithm first got proposed by Debroy and Wong [8]. The concept of MUTREPAIR originates from a mutation technique, that was initially used for assessing the efficacy of a test suite. The mutation technique introduces syntactic code changes into a program and then observes if the test suite is capable of detecting these alterations [8]. The MutRepair algorithm modifies the program flow by introducing mutations to suspicious if conditions, for instance, from == to != or < to <=. The approach considers three kinds of mutations operators: relational, logical and unary [41]. Furthermore, MutRepair performs an exhaustive search [38] on the existing population. It considers all candidates in the current generation and applies each of the predefined mutation operators to each individual candidate, as shown in Figure 2. In comparison, GenProg employs a different approach. Instead of applying all mutations to all candidates, GenProg only applies one randomly selected mutation to a randomly selected subset of candidates [].

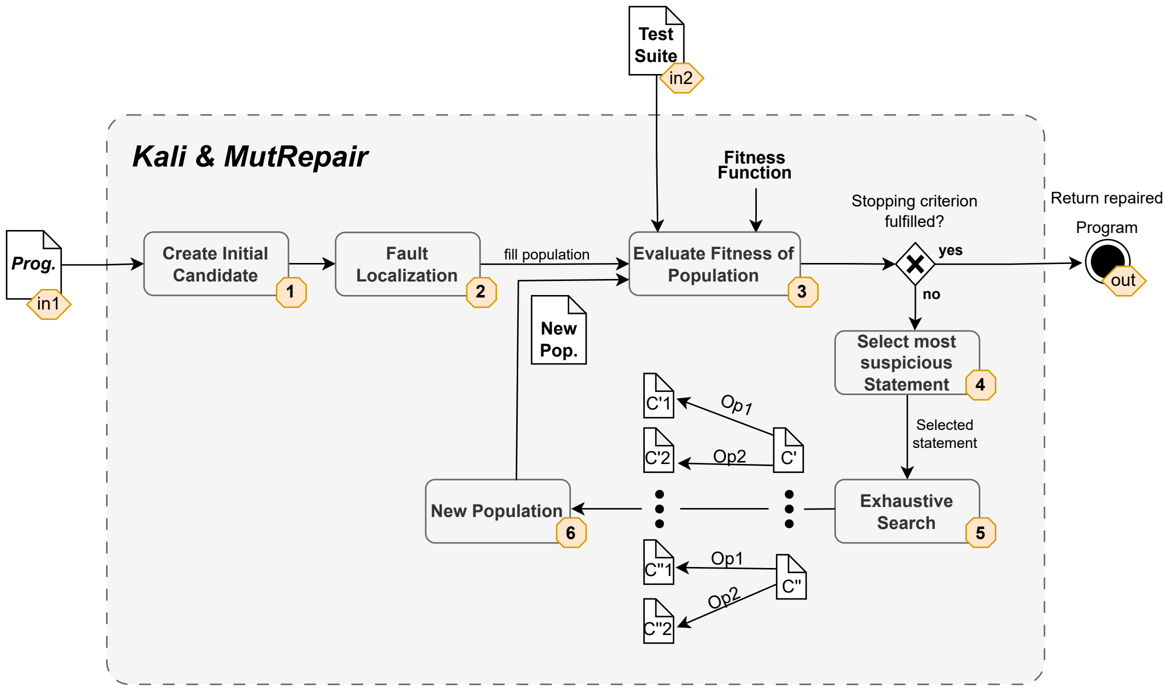


Figure 2: Overview of KALI and MUTREPAIR algorithm

2.4.3 Kali

The KALI program repair approach follows the GENPROG repair loop, but leverages mutation operators that remove or skip certain parts of the code that are considered as the most likely cause of the failing tests based on the fault localization. Similar to MUTREPAIR, KALI utilizes an exhaustive search and applies each mutation operator to each candidate as shown in Figure 2. The purpose of KALI is a bit different to the other approaches. The intent of KALI is not to repair a program, instead it is to find weak test suites [38]. This is because Kali never adds something of value to the code, but instead changes the flow of the program.

2.4.4 Cardumen

CARDUMEN [40] is the most sophisticated of the presented approaches. The goal of CARDUMEN is to find the maximum number of test-suite adequate patches, by extending the repair search space and exploring as many patches as possible. The approach operates on the premise that valuable insights can be gained from every patch identified. Unlike other approaches that impose a fixed generation limit, CARDUMEN instead utilizes a time limit, continuing its search until the allotted time expires while preserving all discovered patches. The core idea of CARDUMEN is based on the competent programmer hypothesis, which suggests that most programmers are competent enough to produce source code that is either correct or only slightly differs from the correct code. Instead of reusing unmodified code like GENPROG, the concept of mining code templates is introduced by CARDUMEN. A code template is a statement in which each variable has been replaced by a placeholder. Code templates themselves are not new and were previously used by PAR [22] and SPR [35]. However, there is a key difference: PAR and SPR use predefined, manually written templates. In contrast, CARDUMEN mines the templates during the repair process from the faulty program. The expansion of the search space is made possible by the introduction of code templates. The repair always consists of a replacement of the suspicious statement by an instance of a template. A template instance represents a template in which the placeholders are substituted with variables that are within the scope of the suspicious statement. The repair process can be divided into two distinct phases: the mining of templates and the navigation of the template space. The initial phase is the mining of templates based on the statements contained within a given program. The templates take a single statement and introduce placeholders for each variable present in that statement. The templates that have been mined are referred to as the template pool. When a template is randomly selected from the template pool for mutation, the placeholders are replaced by arbitrary variables that are present in the scope of the fault location. The collection of template instances can be large. For example, if there are ten variables in Scope and four placeholders in the template, then we create 10,000 template instances (i.e. 10^4). CARDUMEN proposes a solution for this problem. For the second Phase, navigation of the search space, the repair approach creates a probabilistic model based on the occurrence of variable names.

$$\mathrm{pml}_n(\mathrm{v}_1, \ldots, \mathrm{v}_n) = \frac{\text{number of statements containing } (\mathrm{v}_1, \ldots, \mathrm{v}_n)}{\text{all statements with } n \text{ variables}}$$

Sequentially, CARDUMEN creates different entries for the model.
Each entry $\mathrm{pml}_i(\mathrm{v}_1, \ldots, \mathrm{v}_i)$ represents the probability of occurrence of a set of i variables in a statement with i variables. The selection of the template instance, which is then applied at the location of the faulty code, is based on the probabilistic model.

This approach aims to replace the original code with a similar one by selecting templates that closely match the structure of the faulty code. The placeholders within the template are filled based on how frequently the variables co-occur in the program. Both of these design decision are directly rooted in the competent programmer hypothesis.

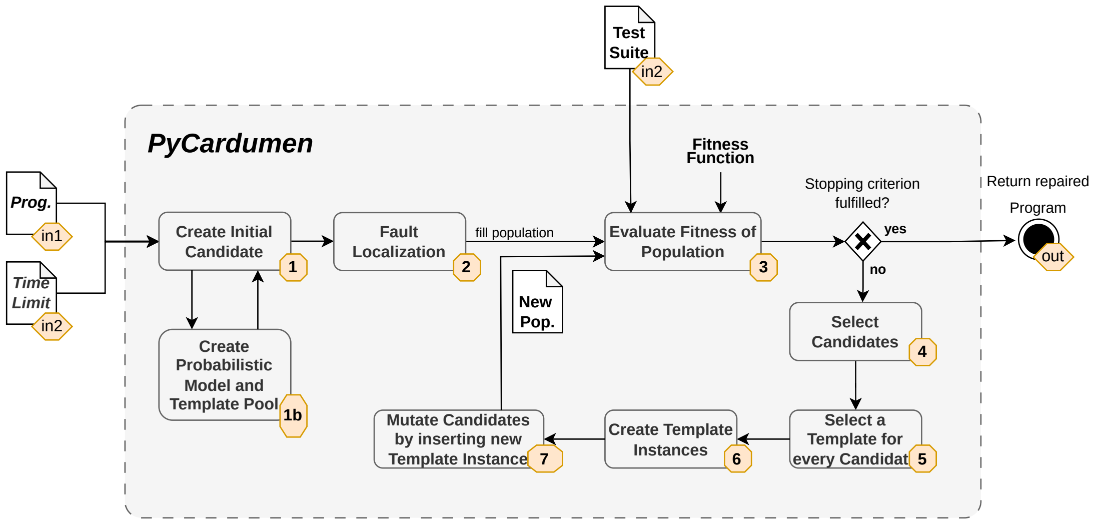

Test Suite
in2
PyCardumen
Fitness Function
Stopping criterion fulfilled?
Return repaired Program
Prog.
in1
Time Limit
in2
Create Initial Candidate
1
Fault Localization
2
fill population
Evaluate Fitness of Population
3
yes
no
out
New Pop.
Select Candidates
4
Create Probabilistic Model and Template Pool
1b
Mutate Candidates by inserting new Template Instance
7
Create Template Instances
6
Select a Template for every Candidat
5

Figure 3: Overview of CARDUMEN algorithm

Background Summary

In this section, we have presented all relevant background information for search-based APR.

In particular, we examined **Genetic Programming** in the context of APR, as well as **Spectra Fault Localization** using the similarity coefficients Tarantula and Ochiai. **Minimisation** also plays an important role in APR, as unnecessary mutations can occur when searching for a patch with APR. We explored how to minimize code while preserving specific properties through example applications of **Delta Debugging**. Finally, we introduced the approaches GENPROG, CARDUMEN, KALI, and MUTREPAIR in detail, as these are implemented in FIXKIT.

3 Implementation of Cardumen in Python

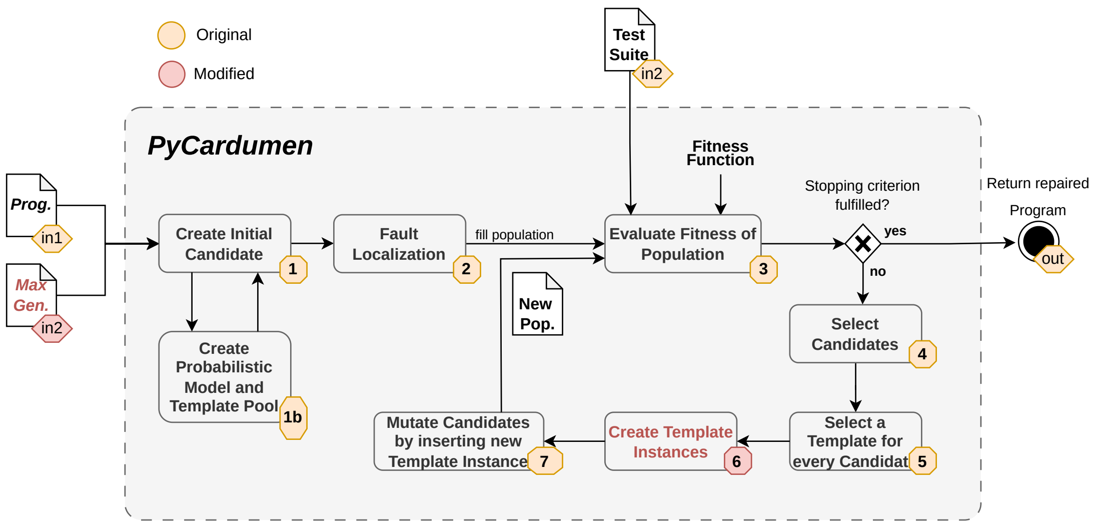
In this section, we examine the modifications to FIXKIT that were necessary for implementing CARDUMEN in Python. These enhancements are summarized in Figure 4. We will have a closer look at the implementation of templates and the probabilistic model (1b). Additionally, we will discuss Template Instance Creation (6), which required optimization to achieve a reasonable runtime in Python. Lastly, we will address a modification that was made to seamlessly integrate PYCARDUMEN into FIXKIT and for ensuring a fair comparison across all approaches in our evaluation.

3.1 Templates

All code mutations in FIXKIT rely on Python's ast (Abstract Syntax Tree) library, including the templates used in PYCARDUMEN. For each statement in the faulty program that we attempt to repair, a template is created. During the initialization of the template class, an AST statement object is passed, which serves as the foundation for the template.

Internally, all variables within the statement are collected. This process is essential as it allows us to store these variables as placeholders, which can later be replaced with new variables. The VarNamesCollector class is responsible for this task. It extends the ast.NodeVisitor class and implements a visit function, enabling it to traverse the tree-like structure of the AST statement. During traversal, it identifies nodes and performs specific operations, such as storing all variables found within the statement.

Additionally, the template maintains a mapping that records the current variable assignments for each node, which can be modified later when instances of the template are created, allowing for flexible adaptation during the repair process.



Original
Modified
Test Suite
in2
PyCardumen
Fitness Function
Stopping criterion fulfilled?
Return repaired Program
Prog.
in1
Max Gen.
in2
Create Initial Candidate
1
Fault Localization
2
fill population
Evaluate Fitness of Population
3
yes
no
out
New Pop.
Select Candidates
4
Create Probabilistic Model and Template Pool
1b
Mutate Candidates by inserting new Template Instance
7
Create Template Instances
6
Select a Template for every Candidate
5

Figure 4: Overview of CARDUMEN algorithm

3.2 Probabilistic Model

Cardumen's mutation operator involves inserting template instances instead of the most suspicious statement. To generate a template instance from a template, a new combination of variables must be assigned to the placeholders. The probabilistic model determines this assignment.

The probabilistic model is implemented using a dictionary, where the key represents a combination of variables, and the value corresponds to the probability of selecting that combination for creating a template instance. The probability is computed using the following equation:

$$\mathrm{pml}_n(\mathrm{v}_1, \ldots, \mathrm{v}_n) = \frac{\text{number of statements containing } (\mathrm{v}_1, \ldots, \mathrm{v}_n)}{\text{all statements with } n \text{ variables}}$$

To compute the probabilities, it is necessary to generate all possible variable combinations. This process involves iterating over the statements of the faulty program, collecting variables using the VarNamesCollector class, and computing all possible combinations using the combinations_with_replacement function from Python's itertools library.

For each variable combination, the probability is calculated based on a predefined equation. This requires determining how many statements contain the exact same variable combination and how many statements contain the same number of variables in general. To achieve this, all statements are iterated over again, comparing their variable sets and variable counts to the generated combinations.

The selection process for assigning variable combinations to template instances follows the principle that the more frequently a variable combination appears, the better suited it is for the template.

3.3 Creation and Selection of Template Instances

In the original Cardumen approach, a template is first selected, and then instances of this template are created for all possible variable combinations. The placeholders within the template are assigned variables from these combinations. Subsequently, a weighted random selection based on the probabilistic model determines the final template instance to be used.

In PYCARDUMEN, a template is still selected, but instead of generating all possible template instances, the probabilistic model is filtered by the number of variables in the selected template. A weighted random selection is then applied to choose a single variable combination, which is used to initialize just one template instance.

This optimization significantly reduces the number of unnecessary template instance creations, making PYCARDUMEN feasible. Without this modification, the approach was computationally too expensive and impractical due to excessive runtime.

3.4 Difference Between Cardumen and PyCardumen

There is a fundamental difference between CARDUMEN and PYCARDUMEN. The original CARDUMEN approach is designed to maximize the number of plausible repairs, as its authors believe that each unique patch provides valuable insights, ultimately contributing to the generation of a correct patch. Consequently, in the original paper, CARDUMEN was executed with a time limit per run rather than a predefined number of iterations.

In contrast, PYCARDUMEN employs an evolutionary search where the process terminates either upon discovering a successful patch or after reaching the maximum number of generations. This modification was introduced to ensure comparability between the approaches implemented in FIXKIT during evaluation, focusing solely on whether PYCARDUMEN can identify a repair, rather than the number of repairs it generates.

However, the original CARDUMEN approach, with a time limit instead of a generation limit, can be seamlessly integrated into FIXKIT if needed.

Implementation Summary

In this section, we had a closer look at the implementation of PYCARDUMEN. Specifically, we focused on the creation of **templates**, which are dynamically generated based on the underlying subject code base. We also discussed the **probabilistic model**, which influences the selection of the template instance used for mutation. Unlike the original CARDUMEN, PYCARDUMEN **first selects the variable assignment** for the template and then creates the template instance. Additionally, instead of using a time limitation to decide when to terminate, PYCARDUMEN implements a **generation-based termination** criterion.

4 Experimental Design

This chapter presents the experimental design of the study, which aims to systematically address the research questions. It outlines the research questions, describes the selected benchmark, and details the experimental procedure to answer the research questions. This structured approach ensures that the results are valid, reproducible, and meaningful.

4.1 Research Questions

This study formulates three key research questions aimed at assessing the effectiveness and applicability of mutation-based automatic program repair for the Python programming language. These questions serve as a guiding framework for the conducted experiments. The analysis of the data generated through these experiments enables a systematic investigation of both the potential and the limitations of the examined approaches.

RQ1: To what extent can the established approaches implemented in FIXKIT find repairs in Python? The first research question examines the fundamental ability of the approaches implemented in FIXKIT to repair errors in Python code. Previous studies have predominantly focused on programming languages such as Java or C, so it is important to assess whether these approaches are also effective for dynamically typed languages such as Python. Due to the inherent differences between dynamically typed languages such as Python and statically typed languages such as Java or C, conceptual challenges may arise.

RQ2: How do the approaches compare to each other in runtime and repairs found? The second research question examines the efficiency and effectiveness of the repair approaches. The focus is on comparing execution times and the number of successfully performed repairs. Additionally, this question aims to differentiate the approaches from one another or identify potential similarities. This analysis is crucial for gaining a deeper understanding of the strengths and weaknesses of each approach, enabling informed decisions about their applicability in different scenarios, and identifying potential optimisation opportunities.

RQ3: Is a genetic approach useful for automatic program repair? Finally, the third research question explores whether genetic programming is a suitable choice for Automatic Program Repair. In order to answer this question, we analyse the evolution of the maximum fitness of the entire population over generations. Additionally, we examine whether there is a tendency regarding when a repair is found.

Answering these research questions provides a comprehensive foundation for evaluating the effectiveness of the repair approaches, determining their suitability for Python, and identifying their limitations.

4.2 Student Assignments

In order to address the research questions, the Refractory benchmark from the University of Singapore will be utilised. This benchmark was developed by a team of researchers specialising in the field of Automatic Program Repair, and it comprises 1,783 distinct subjects derived from student submissions for programming exercises. These subjects are divided into five different 'questions'. Initial experiments with the benchmark revealed that 59 out of the 1,783 subjects were already plausible before any modifications were applied. Each task is accompanied by a problem description, a working reference solution, and a corresponding test suite. The test suite includes 11 test cases for Question 1, 17 for Question 2, 6 for Questions 3 and 4, and 5 for Question 5.

In the first task, students are required to design a function that searches for a specific element in a sequence in linear time. If the element is not found, the function should return the length of the sequence instead. The reference solution consists of five lines of code, while the submitted solutions from participants may vary in length. The second task involves extracting the unique days or months from a given set of birthdays and returning them. The reference implementation consists of 19 lines of code. In the third task, students are required to process a list by removing all duplicate elements. The result should be a new list containing only unique elements. The reference solution accomplishes this in 6 lines of code. The fourth task requires sorting a set of tuples, where each tuple represents a person with gender and age. The goal is to order the tuples in descending order based on age, ensuring that older individuals appear first. Additionally, each age should only appear once in the final output. The reference implementation achieves this in 8 lines of code. The fifth task requires extracting the k largest numbers from a given set and sorting them in descending order. The reference implementation accomplishes this in 6 lines of code.

Question	**Exercise**	**Subjects**	**Test cases**
1	linear search	575	11
2	birthdays	435	17
3	remove duplicates	308	6
4	sorting tuple	357	6
5	extract k-biggest numbers	108	5

Table 4: Overview of the student assignments benchmark

4.3 Experimental Setup

The student assignment benchmark consists of 1,783 different subjects. Since the approaches rely partly on probabilistic decision-making, each approach is executed 10 times on the benchmark to ensure statistically meaningful measurements. While certain studies advocate for 30 repetitions [4, 5], the majority of evaluations in the literature employ only 10 repetitions [26, 40, 47]. With 10 repetitions, this results in 17,830 executions per approach. Given that the FIXKIT implementation supports four approaches, the total number of executions is 71,320. An upper time limit of 1,800 seconds per run is imposed. Should a run not complete within this time frame, it is terminated and marked as invalid, and is not repeated. The following run is then initiated. Consequently, the number of valid data points per approach for our evaluation may be fewer than 17,830.
The evolutionary approaches, PYGENPROG and PYCARDUMEN, are executed with a population size of 40, a maximum of 10 generations, and a mutation probability w_mut of 0.06. The mutation probability w_mut determines whether a suspicious statement undergoes mutation. The exhaustive search approaches are executed with a maximum of one generation. Such parameters are common in the literature when evaluating search-based automatic program repair [26, 47].
During execution, the maximum achieved fitness, the runtime, and whether a repair was found are recorded in a CSV file, which can be accessed in the git repository.

Server The experiments are executed using Slurm on the Gruenau servers. In order to ensure the usability of Slurm while maintaining comparable results, particularly regarding the runtime of the approaches, the execution is restricted to the servers gruenau7 and gruenau8. These two servers are of the same model, equipped with an Intel Xeon 6354 CPU and one terabyte of RAM.

4.4 Research Protocol

Prior to the analysis of the data, it is necessary to clean the data. To this end, the data points from the 59 subjects that were already plausible before the repair are removed. Additionally, data points that did not result in a valid outcome are excluded. An invalid outcome is defined as a run that was interrupted by an exception. In our case, these are primarily Timeout Exceptions, as the run exceeded the $1,800$ second time limit.
In order to address Research Questions 1 and 2, the effectiveness of the approaches in terms of their ability to generate patches will be examined. A statistical hypothesis test will be performed to compare the approaches. Given that more than two approaches are being compared, an ANOVA test will be employed if the data is normally distributed, or alternatively, a Kruskal-Wallis test will be used if the data is not normally distributed. Additionally, we will examine the unique and common fixes of the approaches to compare their search spaces.
In order to address the efficiency aspect of Research Question 2, the approaches will be compared in terms of their runtime. Two additional statistical hypothesis tests will be conducted. Firstly, the total runtime will be compared, examining the times of all valid

runs. Secondly, a subset of the total runtime, the time to fix, will be analysed separately. The time to fix refers to the time required to successfully generate a patch.
In order to evaluate the usefulness of genetic programming in the context of APR, an analysis will be conducted of the generation in which a repair was identified. Additionally, the development of the fitness over the generations will be analysed.

If new questions arise at any point during the evaluation that are necessary to answer the research questions, new exploratory experiments are conducted.

Experimental Design Summary

In this section, the **research questions** were introduced, guiding us toward a comprehensive evaluation of APR in Python. The **student assignment benchmark** and the **parameters** for the experiments were also presented. Finally, the plan for analysing the collected data was outlined.

5 Evaluation and Discussion

In this section, we evaluate the data collected using FIXKIT on the student assignment benchmark. The objective of this study is to answer the following research questions:

> **RQ1:** To what extent can the established approaches implemented in FIXKIT find repairs in Python?
>
> **RQ2:** How do the approaches compare to each other in runtime and repairs found?
>
> **RQ3:** Is a genetic programming approach useful for automatic program repair?

Initially, the repairs generated by the APR approaches are analysed to better understand potential differences between them. To this end, a statistical hypothesis test is performed to determine whether the observed differences in repairs found are statistically significant. To analyse and compare the different search spaces of the approaches, we investigate which ones generate patches for the same subjects. A Venn diagram is used to illustrate these overlaps. Particular attention is paid to the unique patches of an approach, i.e. patches that were only found by one approach and no other.
Apart from the identified repairs, runtime is another crucial factor for APR. To compare the approaches in terms of their execution time, the average and median runtime across all runs are considered. A distinction is made between time to fix and overall runtime.
Finally, the fitness in PYGENPROG is examined in more detail. Additionally, the distribution of the identified repairs across generations is analysed.
Initially, 71,320 executions were attempted, however, 260 of these were terminated without a valid result, indicating that they encountered an exception. In the majority of cases, this was a Timeout Exception, which occurred when the execution exceeded the predefined time limit of 1,800 seconds. Furthermore, 590 runs involving already plausible subjects from the dataset were removed. Consequently, the final dataset comprises 68,700 valid data points, which form the basis for the subsequent evaluation.

5.1 Repairs

The most substantial question regarding Automatic Program Repair is whether it is, at all, finding a repair. A repair is a modification to the source code that ensures the program passes the test suite. The box plot in Figure 5 illustrates the number of repairs found by each approach across all runs. The number of repairs identified by GenProg ranges from 101 in the worst run to 128 in the best run. The average, at 117.1, is slightly lower than the median of 120.0. A greater spread of data below the median can be observed. This is indicated by the lower quartile being larger than the upper quartile and the presence of two lower outliers. Overall, the repairs are strongly centred around the average, with a variance of 58. This suggests that PYGENPROG consistently generates a stable number of repairs, although with a slight tendency towards lower outliers.
Similarly, for PYCARDUMEN, we observe a slightly lower average of 90.7 compared to the median of 93.5, but with a significantly higher spread of values compared to PYGENPROG,

with a variance of 139. However, the values are more evenly distributed across the upper and lower quartiles than in PYGENPROG, except for a strong outlier with 63 repairs found. This suggests that PYCARDUMEN exhibits more variance in the number of repairs found. Since both approaches are based on a genetic algorithm and are highly dependent on probabilities, it is crucial to run them multiple times. Only through repeated runs can a more accurate and realistic picture of their true performance be obtained. In order to provide a comprehensive evaluation of the maximum potential of each approach on the student assignment benchmark, additional runs would likely have been beneficial. However, since the primary focus of this study is to make comparisons between the approaches, the 10 runs selected are deemed to be adequate. This number is consistent with the typical number of executions used in similar studies in the literature [26, 40, 47].

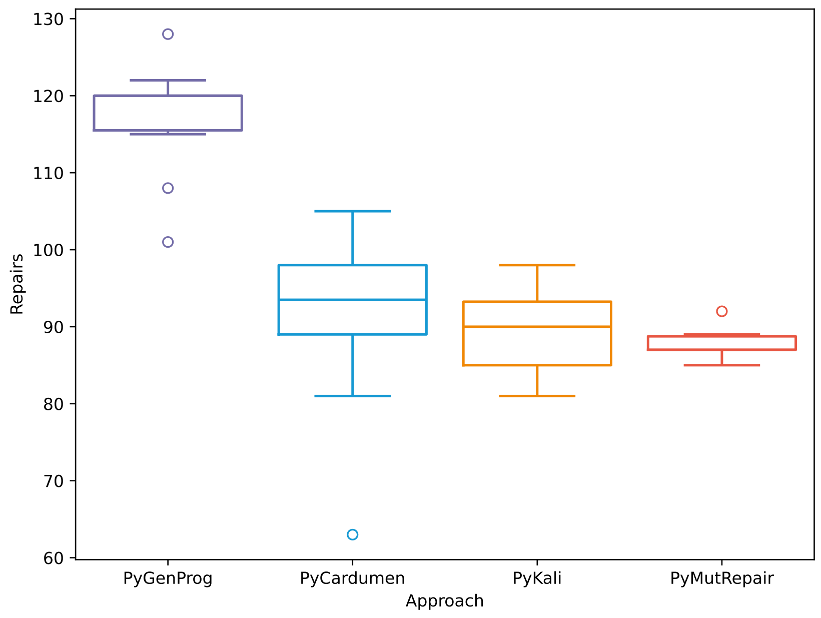


Figure 5: Box plot of the repairs found by the approaches

PYKALI finds a minimum of 81 repairs and a maximum of 98 repairs. The mean is 89.4, and the median is 90.0. With a variance of 28, the differences between the runs are smaller compared to PYGENPROG and PYCARDUMEN. However, the smallest variance between the runs is observed with PYMUTREPAIR. With an average of 87.7 and a median of 87.0, as well as a variance of 4, the maximum number of repairs found, 92, is even depicted as an outlier in the graph in Figure 5. Although the fluctuations in PYMUTREPAIR are limited compared to the other approaches. There should be NO differences at all between the runs in either PYKALI or PYMUTREPAIR. This is due to both approaches using exhaustive search rather than evolutionary search. This means that in each repair attempt, all possible mutations are tested, and if a repair exists within their search space, it should be found each time since the entire search space is explored.

However, this is not the case here, suggesting implementation errors in FIXKIT. More details can be found in the Threats to Validity section 6. The values discussed in this paragraph are summarized again in Table 5.

Statistics	PyGenProg	PyCardumen	PyKali	PyMutRepair
mean	117.1	90.7	89.4	87.7
median	120.0	93.5	90.0	87.0
variance	58	139	28	4
minimum	101	63	81	85
maximum	128	105	98	92

Table 5: Descriptive statistics for each approach, including mean, median, variance, minimum, and maximum values

5.1.1 Statistical Hypothesis Test

In this section, a statistical hypothesis test is conducted to determine whether the apparent differences between the approaches, as shown in Figure 5, are statistically significant. A summary of the test can be found in the appendix 8.1. A statistical test analyses data to determine whether there is significant evidence to support or reject a hypothesis. In our case, we want to know if there are differences in the effectiveness of the approaches. The effectiveness of the approaches is measured in repairs found. The commonly used significance level of $\alpha = 0.05$ is adopted, as recommended in the literature [4, 5]. The significance level α is a predefined threshold that guides the decision to reject the null hypothesis. In other words, it represents the probability of rejecting the null hypothesis when it is in fact true.
Statistical tests compute a p-value, which indicates the probability of obtaining at least as extreme values as those observed in our data, assuming that the null hypothesis is true. If the p-value falls below the predefined significance level, the null hypothesis is rejected. However, this does not imply that the null hypothesis is false. Rather, it suggests that observing such data under the assumption that all test conditions, including the null hypothesis, are true is unlikely [16, 56].
To begin with, the following hypotheses are formulated:

Null hypothesis H_0 There is no difference in the effectiveness of the approaches, i.e. the mean of repairs found is the same for all approaches.

Alternative hypothesis H_1 At least one approach differs significantly from the others in terms of effectiveness.

In order to test these hypotheses, it is necessary to compare all approaches with each other. In cases involving more than two groups, as is the case in this study, either the

ANOVA test or the Kruskal-Wallis test can be applied. The ANOVA test is appropriate if the data are normally distributed, whereas the Kruskal-Wallis test is used for non-normally distributed data.

Checking the prerequisites Before conducting the statistical hypothesis test, the prerequisites for the ANOVA test must be verified. First, the normality of the data distribution is assessed, which can be done either analytically or graphically. In this case, we first test the data analytically using the Shapiro-Wilk test and the Kolmogorov-Smirnov test. For this purpose, the following hypotheses are formulated:

Null hypothesis H_0 The values are normally distributed.

Alternative hypothesis H_1 The values are not normally distributed.

The Shapiro-Wilk test assesses how well the data conforms to a normal distribution. A W-statistic close to 1.0 indicates a strong fit to normality. All approaches achieve a score of 0.87 or higher, suggesting a reasonable but not perfect fit. The p-values are low but do not fall below the predefined significance threshold. The only Exception being PYKALI, whose p-value of 0.85 is significantly higher than those of the other approaches. Additionally, PYKALI exhibits the highest W-statistic among all approaches.

Approach	**W-Statistics**	**P-Value**	**Result**
PyGenProg	0.88	0.15	No Rejection of H_0
PyCardumen	0.87	0.11	No Rejection of H_0
PyKali	0.96	0.15	No Rejection of H_0
PyMutRepair	0.90	0.24	No Rejection of H_0

Table 6: Results of the Shapiro-Wilk Test

The Kolmogorov-Smirnov test can be applied to any distribution and measures the maximum difference between the empirical and theoretical distribution functions, which in this case is the normal distribution. A larger D-statistic indicates a greater deviation from normality, whereas a smaller value suggests a better fit. In this case, all p-values are well above 0.05. However, the D-statistics are not low enough to indicate a good fit between the data and the normal distribution. The only exception is again PYKALI, which achieves a D-statistic of 0.14.

Approach	D-Statistics	P-Value	Result
PyGenProg	0.25	0.49	No Rejection of H_0
PyCardumen	0.24	0.52	No Rejection of H_0
PyKali	0.14	0.96	No Rejection of H_0
PyMutRepair	0.24	0.53	No Rejection of H_0

Table 7: Results of the Kolmogorov-Smirnov test

Since the analytical tests did not yield a clear result, we further examine the data graphically. In the figure, the identified repairs are represented as a probability density histogram, with a normal distribution function overlaid for comparison. In my opinion, the repairs found by PYGENPROG, PYCARDUMEN, and PYMUTREPAIR do not follow a specific distribution but appear to be randomly dispersed. In contrast, PYKALI exhibits the closest resemblance to a normal distribution.

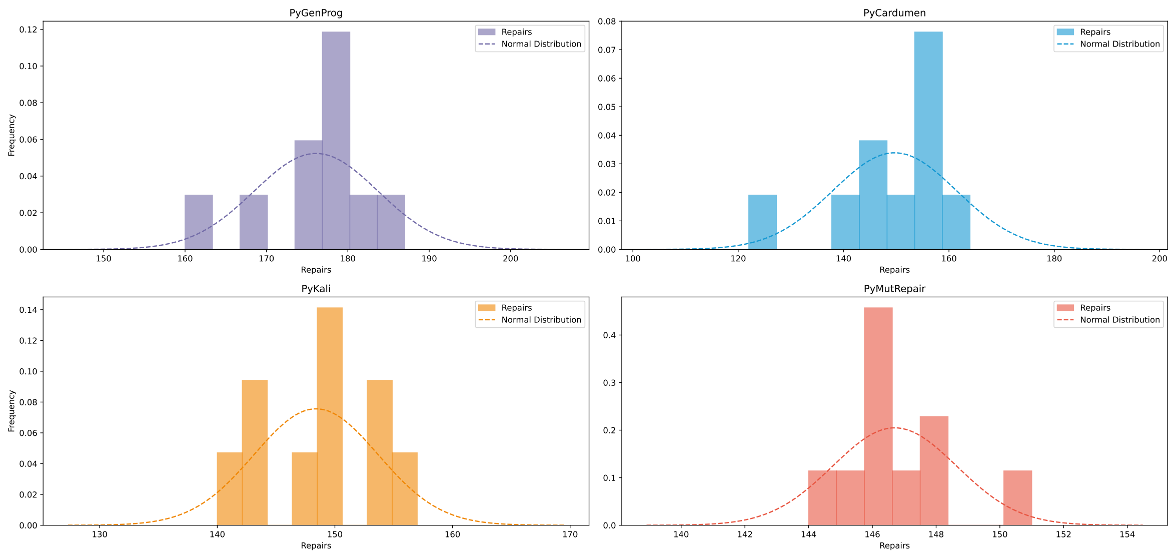


Figure 6: Histogram plots of the probability distribution of the repairs found by FIXKIT

Although the analytical tests did not definitively reject the null hypothesis, their results were not entirely supportive of it either. In conjunction with the graphical representation, we conclude that the data most likely do not follow a normal distribution, with the exception of the data from PYKALI. Consequently, the assumptions for the parametric ANOVA test are not met, and instead, the non-parametric Kruskal-Wallis test must be conducted.

Kruskal-Wallis Test The test resulted in a p-value of 2.46×10^{-5}, which is smaller than the previously set significance level of 0.05. Therefore, the null hypothesis can be rejected. This means that there are significant differences in the number of repairs found between the approaches. To find out which approaches differ significantly from each other, a post-hoc dunn test was performed.
The results show that PYGENPROG differs significantly from all other approaches in terms of the number of repairs found. The mean differences make it clear that PYGENPROG performs significantly better than the other approaches. This is also indicated by the box plot in Figure 5, but the dunn test now provides statistical confirmation. Surprisingly, the dunn test shows no significant difference in effectiveness between the other approaches. Although this was not immediately apparent in the box plot, the dunn test confirms that there are no significant differences between them. We conducted statistical tests to ensure that the observed effects in our data are not due to chance. These tests help differentiate between genuine relationships and random fluctuations, providing an objective basis for our conclusions.

Group Comparison	P-Value	Result	Mean Difference
PyGenProg vs PyCardumen	0.0139	Rejection of H_0	26.4
PyGenProg vs PyKali	0.0008	Rejection of H_0	27.7
PyGenProg vs PyMutRepair	0.00003	Rejection of H_0	29.4
PyCardumen vs PyKali	1.0000	No Rejection of H_0	1.3
PyCardumen vs PyMutRepair	0.7805	No Rejection of H_0	3.0
PyKali vs PyMutRepair	1.0000	No Rejection of H_0	1.7

Table 8: Results of pairwise comparisons of the post-hoc Dunn test

5.1.2 Differences in the Search Space

In the last section, we saw that PYGENPROG finds the most repairs, but does that mean you should only run PYGENPROG and save the computing time for the other approaches? We want to examine this question in more detail in this section.

Before diving into the analysis, let's first clarify some key terms. The table 9 presents both the total patches and the unique patches found. In this context, a patch refers to a subject from the student assignment benchmark that was repaired by a given approach in any run where a repair was successful. This means that if multiple runs of the same approach repair the same subject, it is counted as only one patch. The percentage next to the total patches represents the patch rate, which indicates the proportion of patches found by a specific approach relative to the total number of patches found across all approaches. Unique patches refer to subjects that were only repaired by a single approach and not by any others. The percentage next to the unique patches in the table represents the unique patch rate, which indicates the proportion of unique patches relative to the total number of unique patches across all approaches.

Approach	**Found Patches**	**Unique Patches**
PyGenProg	237 (63%)	57 (30%)
PyCardumen	144 (39%)	53 (28%)
PyKali	145 (39%)	21 (11%)
PyMutRepair	97 (26%)	57 (30%)

Table 9: Overview of the total found and unique patches by every approach.

All approaches together successfully repair a total of 374 patches, resulting in a repair rate of 21.69%. Of these 374 patches, 188 are repaired by only a single approach. The overlaps between the approaches in terms of the patches they have repaired are visualized in Figure 7. Among the individual approaches, PYGENPROG achieved the highest number with 237 patches. PYKALI and PYCARDUMEN show almost the same results with 145 and 144 patches respectively, while PYMUTREPAIR provides the lowest number of repairs with 97 patches.

An interesting aspect is the consideration of unique patches, i.e. patches that were only found by a single approach. Two striking observations can be made here:

PyKali finds the lowest number of unique patches with 21 unique patches, which corresponds to a unique patch rate of 11%. The high overlap of PYKALI, particularly with PYGENPROG and PYCARDUMEN, can be observed in Figure 7. The largest exclusive overlap, referring to patches that were repaired by exactly two approaches and no others, is found between PYGENPROG and PYKALI, with 74 patches being repaired by both approaches. One possible explanation for this could be that both approaches use the delete a statement operator as a mutation operator. To test this hypothesis, PYGENPROG was run again on the 74 subjects that were repaired exclusively by these two approaches — but this time without the delete mutation operator. Surprisingly, PYGENPROG was still able to successfully generate a patch for 71 of the 74 subjects. This shows that the common mutation operator is not the decisive reason for the high overlap of the patches found. It should be emphasised at this point that two approaches that find a patch for the same subject do not necessarily generate identical patches.

PyMutRepair, PYGENPROG and PYCARDUMEN find a similar amount of unique patches, with a unique patch rate of around 30%. An analysis of the Venn diagram reveals minimal overlap between PYMUTREPAIR and any of the other approaches. Specifically, PYGENPROG and PYMUTREPAIR share 38 patches, while PYCARDUMEN and PYMUTREPAIR have 19 patches in common. In contrast, PYKALI and PYMUTREPAIR exhibit the least overlap, sharing only 4 patches. The high proportion of unique patches and the limited overlap with other approaches can be partly attributed to the relatively low total number of patches identified by PYMUTREPAIR. Nevertheless, these observations suggest that PYMUTREPAIR explores a largely distinct search space compared to the other approaches. This clear difference can be explained by the specific way PYMUTREPAIR works. While the other approaches generate patches by deleting, replacing or inserting statements - with PYCARDUMEN also inserting templates from the faulty programs - PYMUTREPAIR takes a different approach. It transforms existing conditions within if statements, which leads to a change in the control flow. For example, a condition of the form if $x < 10$ is converted to if $x <= 10$. Such faults can only be addressed by other approaches if the modified condition (e.g., if $x <= 10$) already exists somewhere in the program. Since this is typically not the case, PYMUTREPAIR can effectively address a specific category of faults, as clearly reflected in the results.

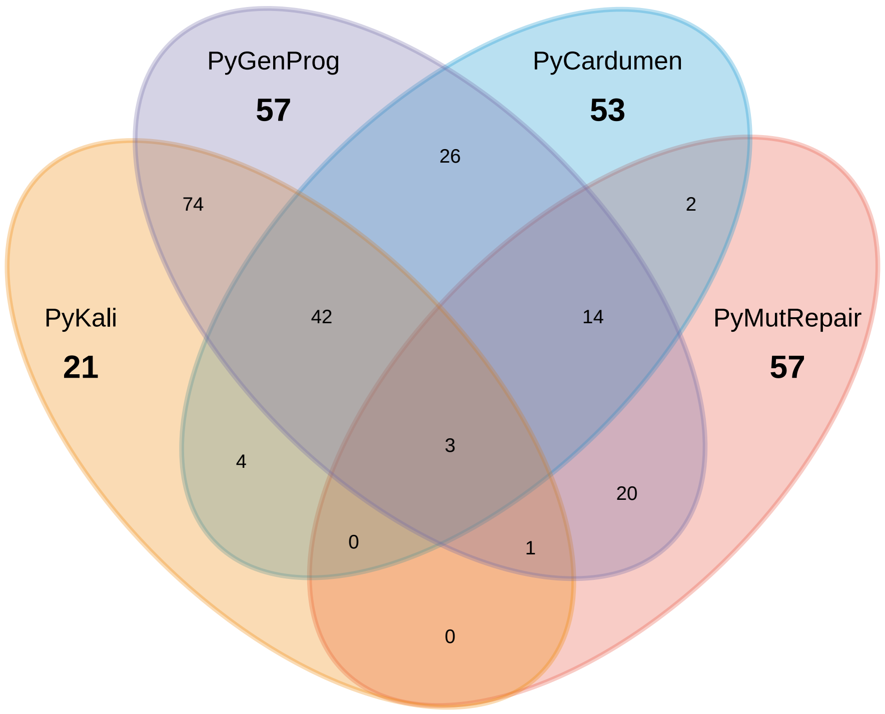


Figure 7: Venn Diagram of all repairs found by FIXKIT

5.2 Runtime

In this section, a more detailed examination of the runtimes of the various approaches is presented. Initially, it is evident from Table 10 that there is a substantial difference between the average and median values of an approach, whether considering runtime or time to fix. An observation of Figures 8 and Figure 9 reveals a considerable number of outliers. A number of factors may be responsible for the variations in runtime, including:

- Differences in the size of the subjects
- The specific working mechanisms of the approaches, which are dependent on the structure of the subject
- Server load variations

The differences in subject size are relatively small, as all subjects contain fewer than 100 lines of code. The runtime of PYCARDUMEN and PYMUTREPAIR depends on the structure of the subjects. For PYCARDUMEN, the runtime is influenced by the number of variables in an individual statement and in the scope. The more variables present, the more variable combinations PYCARDUMEN must compute for the templates. On the other hand, PYMUTREPAIR's runtime is directly impacted by the number of if-statements in the subject, as it only performs transformations on if-statements.
However, the most plausible explanation for the high variance in runtimes appears to be the variable server load during the experiments. The experiments were executed using slurm on the gruenau cluster, a factor that is likely to have had an impact on the runtimes. Further details can be found in the Threats to Validity section 6.
These factors should be taken into account when evaluating the validity of this analysis. Nevertheless, the credibility of the analysis is supported by the large dataset of 68,700 data points, which should at least reflect the general trends of the approaches under investigation.

Metric	**PyGenProg**	**PyCardumen**	**PyKali**	**PyMutRepair**
average runtime	99.7	559.8	38.4	57.8
median runtime	69.6	443.7	28.6	38.9
average time to fix	33.4	136.5	20.1	63.2
median time to fix	13.8	70.5	12.3	47.9

Table 10: Overview of the runtime and time to fix by each approach

Comparison of Approaches Based on Runtime Now let us compare the approaches in terms of their runtime. Runtime is defined as the total execution time required by the program, regardless of whether a repair was successfully found or not. The recorded

times include both successful and unsuccessful repair attempts.
To analyse the runtimes of the approaches, a statistical hypothesis test is conducted once again. In contrast to the previous chapter, this section will not focus on individual values but will instead provide a summary of the test results. All detailed values can be found in the appendix 8.1.
First, the runtimes of the approaches are examined for normal distribution. The tests indicate that the data for all approaches do not follow a normal distribution. Therefore, the non-parametric Kruskal-Wallis test must be applied. This test assesses whether independent samples originate from the same population. If this assumption holds, there is no significant difference between the groups. The null hypothesis tested by the Kruskal-Wallis test is as follows: The runtimes of the different approaches do not differ significantly.
The Kruskal-Wallis test showed with a p-value of 0 that at least one approach differs significantly from the others in terms of runtime. A subsequent post-hoc analysis using the Dunn test shows that all approaches differ significantly from each other.
The data presented in the table 10 clearly show that the exhaustive search approaches have both lower average and median runtimes when compared to the evolutionary approaches. According to the literature, the validation of candidates is known to be the most time-consuming factor [57].
A plausible explanation for this observation is the relatively small size of the subjects. It is faster to explore all possible combinations exhaustively than to run ten generations of the evolutionary algorithm. With the chosen parameters of ten generations and a maximum population size of 40, this amounts to validating a total of 400 candidates.
On the other hand, PYKALI employs eight mutation operators, which are applied to all suspicious statements. A subject would need to contain 50 statements for PYKALI to generate the same number of candidates as PYGENPROG. Given the relatively small size of the programs under consideration, this is rather unlikely. A similar behaviour is observed in PYMUTREPAIR, which uses ten mutation operators, though these are exclusively applied to if-statements.
To verify whether the generated candidates might be the cause of the differences in runtime, a sample of 50 subjects from the benchmark was used to measure the candidates generated by PYKALI and PYMUTREPAIR. The measurements revealed that PYKALI generates an average of 53.6 candidates, while PYMUTREPAIR produces an average of 66.75 candidates. Both values are significantly lower than the 400 candidates generated by the evolutionary approaches.
The comparatively long runtimes of PYCARDUMEN could be due to the fact that the subjects and test suites are relatively small, which means that the precomputing tasks are more significant. However, it should be noted that I unfortunately have no specific measurements of how much time PYCARDUMEN used on which task, so this is only an assumption.

Comparison of Approaches Based on Time to Fix In this section, we focus exclusively on the time to fix for each approach, which is defined as the runtime required to

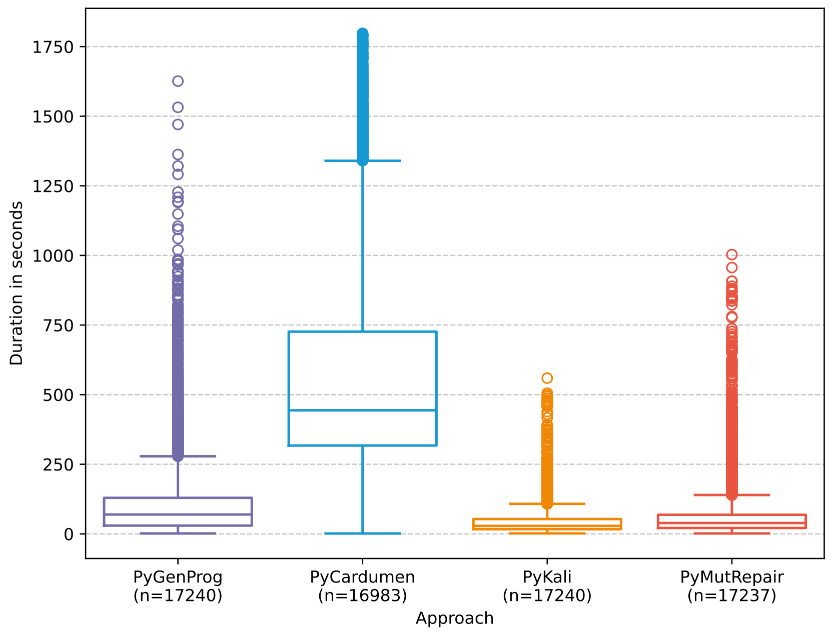


Figure 8: the runtime for each approach

find a repair. All detailed values of the statistical hypothesis test can be found in the appendix 8.1. The time to fix data is a subset of the runtime data and, similarly, does not follow a normal distribution. The Kruskal-Wallis test, with a p-value of 0, confirmed that at least the time to fix of one approach differs significantly from the others. A subsequent post-hoc Dunn test revealed that all approaches differ significantly from one another. A significant difference between PYGENPROG and PYKALI was not immediately apparent when examining the medians in Table 10. However, this was confirmed through statistical testing. A comparison of the runtime in Figure 8 with the Time to Fix in Figure 9 reveals a convergence of the times across the different approaches. In addition to the p-value, the Kruskal-Wallis test calculates an H-value, which measures differences in the rank distribution of the datasets. For the analysis of rank distribution, the data points from the samples are first pooled together, and then each value is assigned a rank, with the smallest value receiving rank 1, the second smallest value receiving rank 2, and so on. The ranks of the values within each sample are then analysed for differences. A higher H-value indicates a more uneven rank distribution among the data points. The H-value for the time to fix is $1,501.68$, which is significantly lower than the H-value of $37,257.37$ for the runtime.

This convergence can be explained by the fact that when one of the evolutionary approaches finds a patch, the repair process is successfully terminated, and the remaining generations are no longer executed. In other words, if a repair is found in the first generation, only 40 candidates need to be created and validated, which is fewer than the observed created number of candidates in the exhaustive approaches. How often this is the case will be analysed in the next section 5.3.

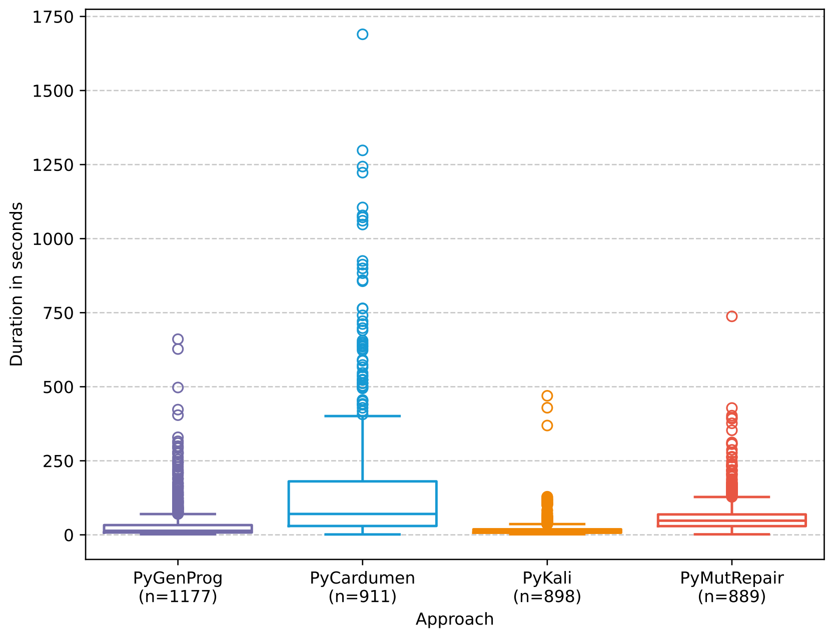


Figure 9: The time to fix for each approach

5.3 Fitness

In this section, we aim to analyse the fitness metric, which is crucial for genetic programming. First, we will examine how the generated patches are distributed across generations i.e., how many patches were found in Generation 1, Generation 2, and so on. Subsequently, we will track the development of the maximum fitness across generations within the population.

Generations to Fix In the bar chart in Figure 10, there is a clear visual concentration of the found patches in the first four generations. There are 641 patches found within the first four generations. In the repairs section, the term "patches" refers to the different subjects for which a repair was successfully found. In this section, however, all runs that resulted in a fitness of 1.0 are counted as patches. Therefore, if a subject had a patch found across three different runs (not necessarily in the same generation), each of these runs is counted as a separate patch.

In the bar chart, the number of patches found decreases across generations, with two exceptions in generations 6 and 7. A clear downward trend is observable.

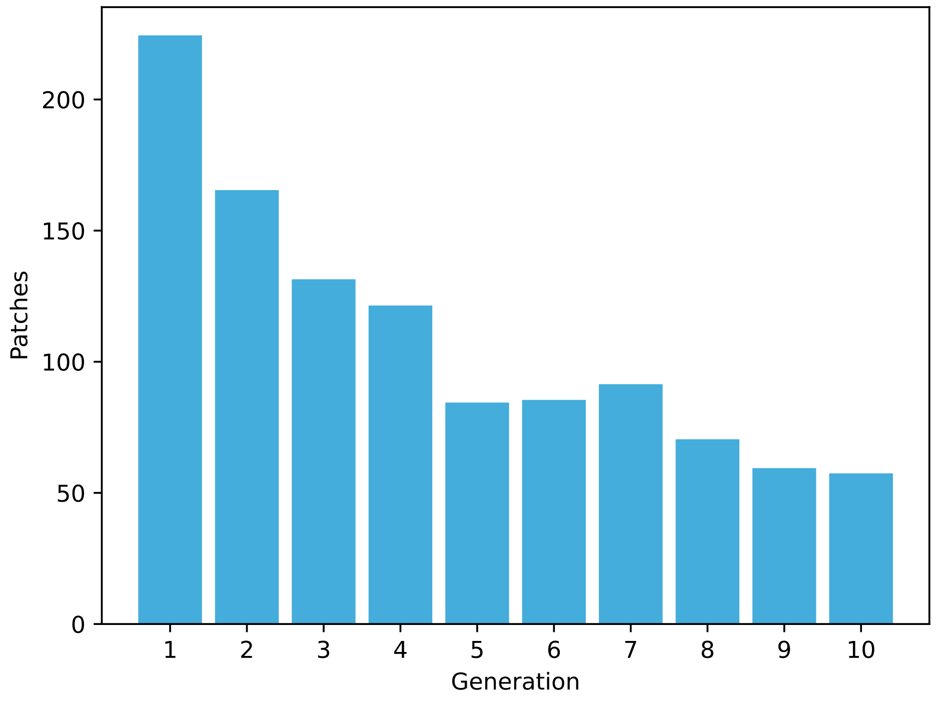


Figure 10: The generations GENPROG found a patch

Evolution of the Fitness Over Generations In this section, we examine the evolution of fitness across generations in PYGENPROG. Each generation consists of 40 candidates, and the maximum fitness achieved by the fittest candidate in each generation was recorded. Figures 11 and 12 illustrate the fitness progression of the fittest candidate per generation for a randomly selected sample of 10 subjects. To maintain clarity and avoid visual clutter, only 10 runs were randomly chosen for this representation, as displaying all runs would have been overly complex.

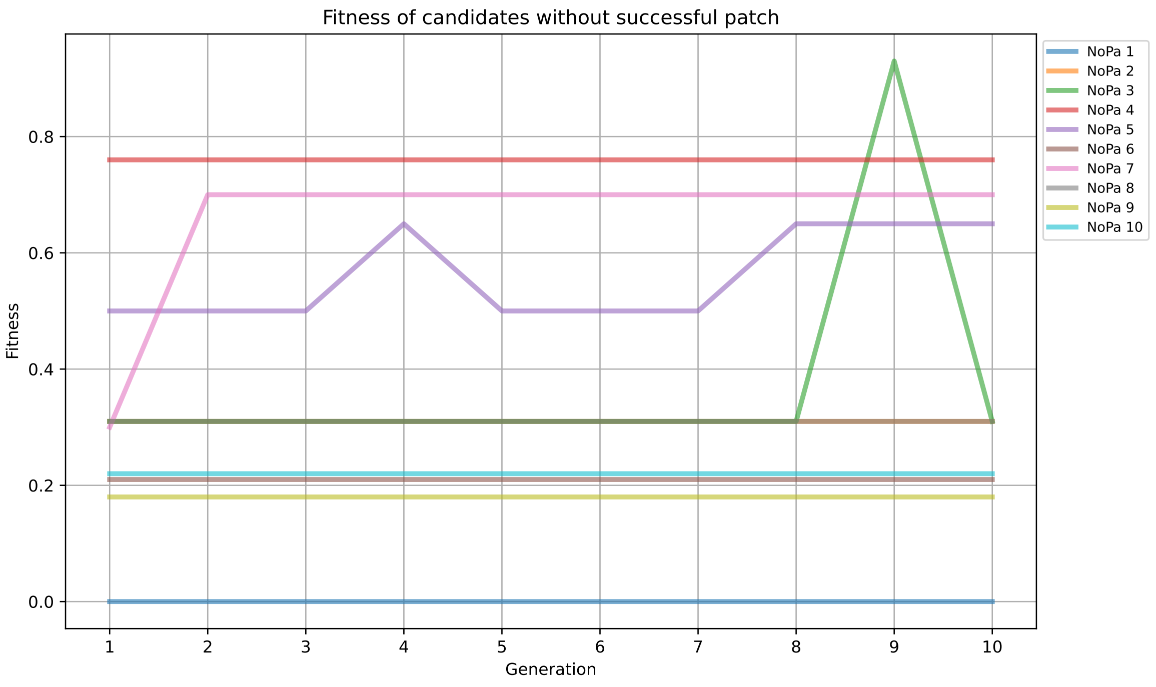


Figure 11: the line plot of the fitness values of some subjects over the generations

Figure 11 presents runs in which no patch was found, labelled as NoPa short for “No Patch”. A stagnation in the maximum fitness across all 10 generations is observed in 7 out of 10 runs. In the remaining 3 runs, at least two distinct fitness values are recorded. The occasional decline in fitness to a lower value is expected and intentional, resulting from the selection mechanism used for advancing candidates to the next generation. Specifically, the roulette wheel selection method, introduced in the genetic programming section, was applied in these experiments. These fitness drops serve to prevent premature convergence and ensure a more diverse exploration of the search space. Notably, significant jumps in fitness following mutations can be seen in runs $NoPa3$ and $NoPa7$. These abrupt increases suggest the occurrence of random “lucky hits” rather than the gradual fitness improvement typically associated with genetic programming. A similar pattern can be observed in Figure 12, which focuses exclusively on runs where a patch was successfully found. Patches 2, 5, 8, and 9 exhibit abrupt jumps in fitness, increasing from 0.3 to 1.0 following a single mutation. This sudden improvement suggests the presence of random, favourable mutations rather than a steady, progressive optimization process. An exception to this pattern is observed in Patch 6, which displays five distinct maximum fitness values, with fitness increasing relatively consistently over generations. This gradual progression aligns more closely with the expected behaviour of genetic programming, where incremental improvements are typically anticipated.

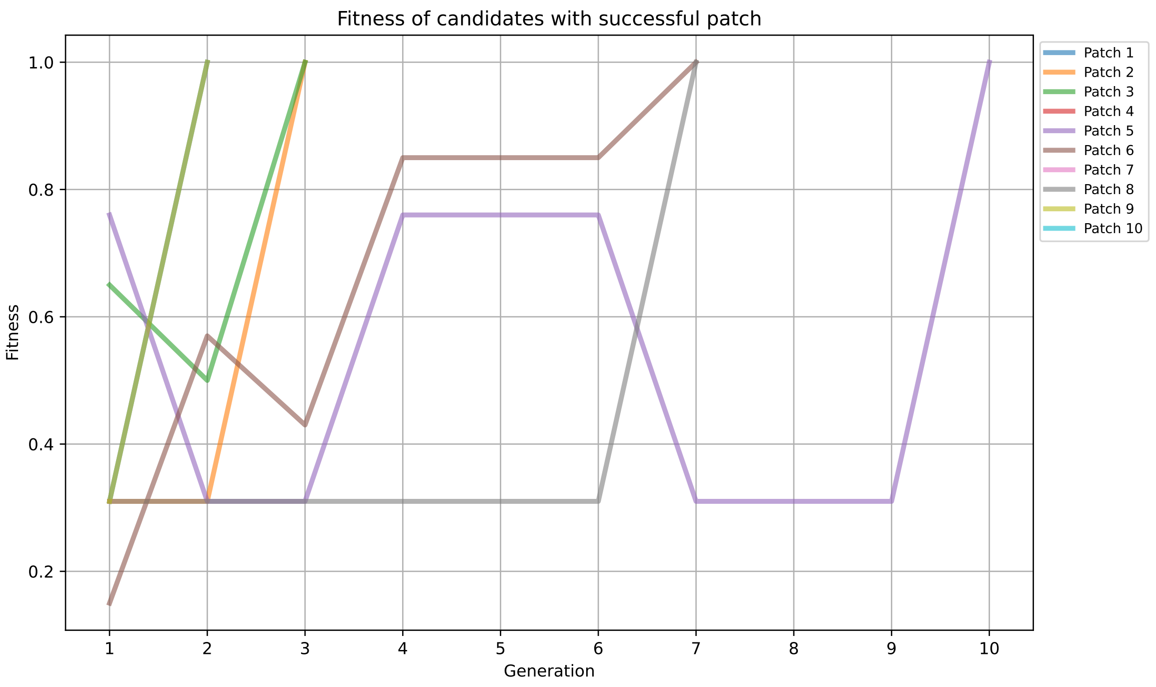


Figure 12: the line plot of the fitness values of some successful repairs over the generations

To determine whether these observations are merely coincidental or provide a realistic representation of genetic programming in the context of Automatic Program Repair, Figure 13 illustrates the unique values that the maximum fitness achieves across generations. The data is presented in two separate bar charts: one for all runs and another exclusively for runs that resulted in a successful patch. The left bar chart of Figure 13 illustrates the changes in maximum fitness across all 17,240 runs, with nearly 10,000 of these runs showing no change in maximum fitness across all 10 Generations, suggesting that the fitness remained consistent with that of the initial subject. The frequency of different maximum fitness values decreases significantly as the number of distinct values increases. The right bar chart in Figure 13 shows the unique values of maximum fitness for successful repairs. One of the fitness values is always 1.0, as this indicates that a patch has been found. This means that the runs with only one maximum fitness value have a fitness of 1.0, which is the only value they attain, indicating that these runs found a patch in the first generation. In Figure 13, we observe just over 200 runs with a single maximum fitness value, which aligns perfectly with the number of patches found after the first generation, as shown in Figure 10. Conversely, the runs with two distinct maximum fitness values initially had a value below 1.0 and then jumped to 1.0 after a specific mutation, similar to Patch 8 in Figure 12. Only from three unique fitness values onwards can we begin to observe the behaviour expected from genetic programming.
The dysfunctional behaviour of genetic programming, as evidenced by the fitness progression of certain subjects, is further confirmed by the fact that 80% of the runs show either no change or only a single change in maximum fitness. The data suggest that genetic

programming for APR does not achieve the anticipated effect of a gradual improvement in fitness, ultimately leading to a successful repair. Instead, it appears more like a waiting game for the one correct mutation that results in the desired behaviour. If this assumption holds, fitness evaluations could potentially be omitted, as they are among the most computationally expensive operations.

There are several reasons why our findings on genetic programming in the context of automatic program repair may not be generalizable. First, the test subjects used in our study are relatively small, which means that the search space for PYGENPROG is not optimal. Since GENPROG relies on the competent programmer hypothesis, this limitation could have negatively affected our results. Second, the effectiveness of genetic programming is highly dependent on the fitness function, which in our case is the test suite. If certain functionalities are tested by multiple test cases, they are automatically weighted more heavily than those that are less extensively tested. Consequently, the quality of the test suite plays a crucial role in shaping the results. A potentially inadequate test suite in the student assignment benchmark may have introduced biases or negatively influenced our findings. Both of these issues are addressable and should be further investigated in future research.

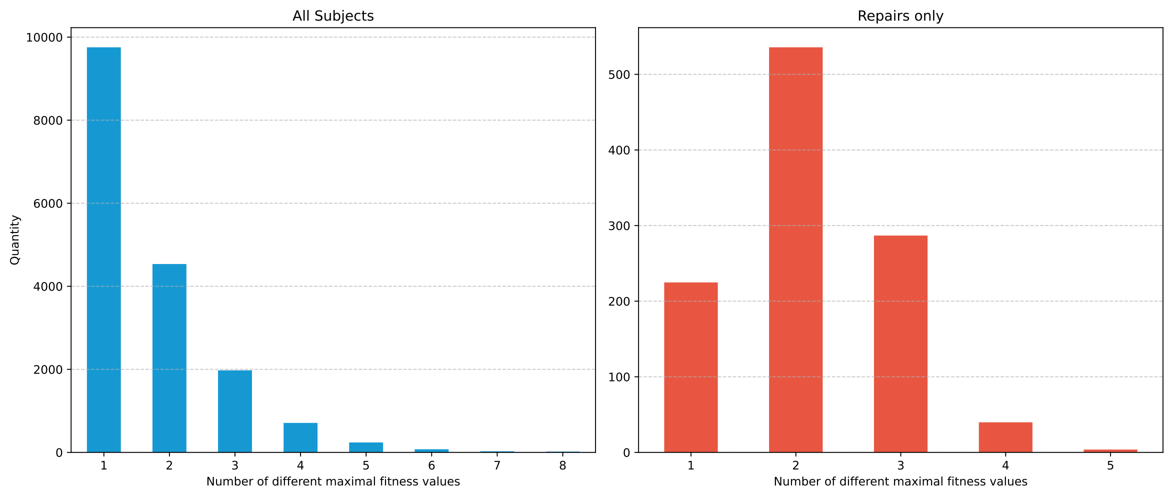


Figure 13: The left bar chart shows distinct maximal fitness values per subject, the right shows the values only from repairs

5.4 Summarizing our Findings

During our evaluation, we made several insightful observations:

1. PYGENPROG finds the most repairs within a reasonable time frame.
2. There is no significant difference in the number of repairs found between PYCARDUMEN, PYKALI, and PYMUTREPAIR.
3. The unique repairs are evenly distributed among PYGENPROG, PYCARDUMEN, and PYMUTREPAIR, suggesting that a diverse collection of APR approaches is beneficial.
4. PYCARDUMEN faces challenges with computational time.
5. PYGENPROG discovers the most repairs in the early generations.
6. Genetic programming does not work as smoothly for APR as initially expected.

These findings provide a comprehensive overview of the strengths and weaknesses of the evaluated approaches and the impact of genetic programming for APR. Building upon this, we now address the specific research questions.

RQ1: To what extent can the established approaches implemented in FIXKIT find repairs in Python? The established approaches in FIXKIT demonstrate a notable ability to find repairs in Python. Specifically, **the approaches achieve a Repair Rate of 21.69% on the student assignment benchmark**. This indicates that a substantial proportion of the subjects within the benchmark can be repaired by the approaches in FIXKIT, but there is still considerable room for improvement.

RQ2: How do the approaches compare to each other in runtime and repairs found? **PYGENPROG outperforms the other approaches overall**, finding the most repairs within a reliable time frame. **The approaches PYGENPROG, PYCARDUMEN, and PYMUTREPAIR exhibit evenly distributed unique repairs**, highlighting their value as complementary components within FIXKIT. If computing time is a critical factor, using PYGENPROG and PYMUTREPAIR may yield the quickest results when looking for fixes.

RQ3: Is a genetic programming approach useful for automatic program repair? While genetic programming can be effective, **its contributions to automatic program repair remain uncertain**. As seen in Figure 13, fitness stagnates in most cases, and **significant jumps in fitness are typically observed after finding the correct mutation, resembling the outcome of a "lucky hit"**, as shown in Figure 11 and 12. In most cases, the patches are found in the early generations, suggesting that the genetic programming is not being used properly. Additionally, no significant time advantage is gained over the exhaustive search approaches, likely due to the small size of the subjects involved. However, genetic programming does offer a theoretical advantage in repairing multiple bugs simultaneously, such as those requiring mutations at multiple locations or different bugs within the same project. The likelihood of hitting the correct mutations, however, is relatively low. Thus, optimizations are needed for genetic programming to be a practical solution for APR. In its current form, the approach does not sufficiently justify the high costs associated with candidate validation.

6 Threats to Validity

In this section, we discuss the several challenges we encountered during our work. The main threats to the validity of our findings fall into three categories: internal validity, external validity, and construct validity.

Internal Threats to Validity One issue that stood out during evaluation was the variability in multiple runs of the exhaustive search approaches. Despite investigation, the exact cause of this inconsistency remains unknown.
Another issue pertains to candidate validation. We discovered late in the process that, in some runs, the 59 already functional subjects were not always correctly identified before the repair process started, indicating a validation issue. Despite investigation, the root cause remains unclear.
Another challenge was the lack of reproducibility in experiment runs, even when using a fixed seed for Python's randomization. This issue had multiple causes. First, there was an inconsistency in the order of suggestions returned by SFLKIT. These suggestions represent the most likely faulty locations where mutations should be applied. This inconsistency was resolved by sorting the suggestions, ensuring a deterministic order across runs. Second, FIXKIT assigns a numerical identifier to each statement for mutation purposes. However, this numbering was dependent on the order in which files were processed. On the Gruenau servers, this file order was inconsistent, leading to variations in statement numbering across runs. This issue was mitigated by sorting the files in lexicographical order, ensuring a consistent numbering scheme. Although these issues were mitigated, we were unable to rerun the experiments due to time constraints.
Additionally, the runtime measurements appear to have been significantly influenced by server load. As a result, the timing data used in our evaluation should be interpreted with caution.
We made every effort to implement all approaches as closely as possible to their original papers, but there is always room for error.

External Threats to Validity The subjects in the student assignments benchmark are relatively small, each containing fewer than 50 lines of code. In contrast, real-world programs often span tens of thousands of lines. This limitation restricts the search space for PYGENPROG and PYCARDUMEN, thereby reducing the likelihood of finding a valid repair. Therefore, our findings may not directly generalize to large-scale real-world applications.
A potential limitation is whether student assignments adequately represent real-world APR challenges, particularly given that some approaches assume the competent programmer hypothesis.
PYKALI is designed to expose weak test suites rather than actively repair programs. However, it still found a significant number of repairs in the student assignment benchmark, suggesting that the test suites in this benchmark may not be comprehensive enough for all edge cases.

We did not test repairs for bugs spanning multiple files, but FIXKIT has the fundamental implementation to handle multi-file bug fixes.
Our experiments were conducted with 10 runs per subject, which is standard in research. However, some studies recommend up to 30 runs to achieve more stable results. Additional runs could provide a more accurate and reliable evaluation by reducing the influence of outliers. With only 10 runs, outliers have a stronger impact on the overall results, whereas a higher number of runs would make the findings more robust against such anomalies.

Construct Validity Threats A known issue in APR is the generation of plausible patches rather than truly correct ones. In our study, we did not manually verify whether the generated patches were genuinely correct or merely passed the given test cases. Based on existing research, it is likely that many of them are only plausible fixes rather than actual corrections. Due to the high number of generated patches, manual inspection was impractical. However, a larger and more comprehensive test suite could help reduce the risk of plausible but incorrect patches.
In our evaluation, we analysed the development of the maximum fitness over generations. However, not all candidates in a generation necessarily have the same fitness, so some will have a lower fitness than the maximum fitness we measured. This means that when a repair was found, it might not have originated from the fittest candidate at that time. The probability of this happening is low, but it remains a possibility. A better approach would have been to track the fitness of each individual candidate, rather than only the maximum fitness of the population. That said, if the repairs did not stem from the previously fittest candidate, the observed jumps in fitness would be even more extreme, further reinforcing our conclusions.
A more granular measurement of runtime - differentiating precomputing, fault localization, mutation, and validation - would have yielded deeper insights than reporting only total execution time.
Finally, while the conclusions drawn from the statistical tests are our own subjective interpretation, we have consistently provided p-values to allow others to form their own interpretations of the results.

Threats of Validity Summary

The biggest limitations are the small subject size, the small test suite, and the bugs in FIXKIT. All of these issues can be addressed through further research and improvements to the codebase.

7 Related Work

This section provides an overview of the related work in the field of automatic program repair, as well as the metrics considered in this context.
The related work is structured according to the different areas of Automatic Program Repair that have emerged over time. The field of automatic program repair can be broadly categorized into two major subfields, as shown in 14: behavioural and state-based repair. Within behavioural repair, the evolution of methodologies highlights distinct phases. It began with search-based approaches, which laid the groundwork for the field. Over time, these were followed by pattern-based and semantic-based approaches. More recently, behavioural automatic program repair expanded to include learning-based approaches driven by new advancements in the field of machine learning.
Subsequently, the metrics used in the literature to evaluate the effectiveness and efficiency of automatic program repair approaches are introduced. In particular, these include repair rate, the correctness of generated patches, repair time, the number of test case executions, and readability.

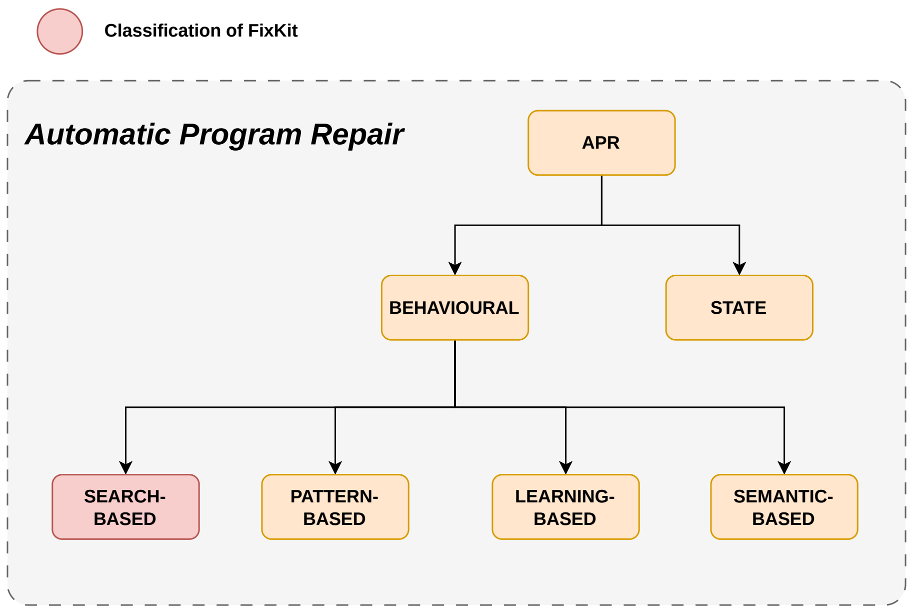


Figure 14: Overview of Automatic Program Repair

7.1 Behavioural Automatic Program Repair

Behavioural automatic program repair involves the modification of the underlying code of a program to eliminate faults.

Search-Based Automatic Program Repair generates candidates and validates them against a test suite. Search-based approaches are often referred to as Generate-and-Validate [63], or heuristic-based approaches [6, 64].
The pioneer and most prominent example, GENPROG [27], employs genetic programming to guide the candidate creation process. For patch generation, GENPROG utilizes mutation operators such as inserting a statement, replacing one statement with another, or deleting a statement.
Similarly, CARDUMEN [40] also leverages genetic programming but dynamically generates templates to expand the search space. The core philosophy of CARDUMEN is to generate a large volume of patches, based on the assumption that each patch provides valuable insights into potential repairs.
However, the effectiveness of genetic programming in the context of automatic program repair remains debated [47], leading to variations in search strategies. For instance, RSRepair [47] employs random search instead of genetic programming.
In contrast, other approaches utilize an exhaustive search strategy to systematically explore the entire search space of their mutation operators [8, 48]. This method ensures that every possible mutation is considered, leaving no potential patch unexplored.
AE [57] aims to reduce the cost of repair by clustering similar candidates to minimize additional test case executions. Instead of testing every candidate individually, AE executes test cases only once per cluster. Additionally, it uses test case ordering, prioritizing those most likely to fail and executing them first.
CapGen [61] utilizes the same mutation operators as GENPROG. In the process of mutating a chosen code statement, CapGen considers the similarity between the context of the fault location and other statements to identify the most suitable statements for generating a patch.

Pattern-Based Automatic Program Repair also referred to as template-based approaches [64], categorize errors into distinct fault categories and provide templates to repair each type of error. Initially, these templates were manually created by researchers [22, 35, 54]. In order to solve a given bug type, the manual creation of templates by a programmer is required. This proved to be an initial large cost.
For example, to prevent null pointer dereferencing, a check for null pointers would be inserted before using an object. Manual template generation has been applied in several approaches, such as PAR [22], Relifix [54], and SPR [35]. However, this approach is not scalable, because of the aforementioned large workload.
This led to the next step: automating the generation of templates. Genesis [33] introduced an approach that automatically infers code transformations for patch generation from sample programs. A limitation of Genesis was the restriction of the search space to three fault categories: null pointer, out-of-bounds, and class casting defects.
This limitation was later addressed by SOFix [32] through the mining of fixing patterns. These patterns were found in posts on the website stack overflow to broaden the scope of repair. These patterns are represented as GumTree edit scripts. Edit Scripts capture the differences required to transform Code Snippet A into Code Snippet B. GumTree is

an algorithm that displays the code differences directly at a node of the abstract syntax tree [12].
REVISAR [49] automatically searches git repositories and their version histories for quick fixes. Quick fixes are code patterns used by static analysis tools to address common errors. REVISAR extracts these patterns and clusters them based on their similarity.
FixMiner [24] combines and extends the concepts of previous approaches, by introducing rich edit scripts to represent fix patterns. These edit scripts include not only the code changes themselves, but also the code context of the mined patterns. These Rich Edit Scripts are paired with a specialized clustering method. In multiple iterations, the edit scripts are first clustered based on identical abstract syntax trees, followed by similar edit action trees, and finally incorporating the code context tree into the clustering process.

Learning-Based Automatic Program Repair In the early stages of Automatic Program Repair, mutations were applied exclusively to the existing code in the faulty program. However, this approach revealed that the search space was not optimal. To create a more effective search space, research shifted towards the use of fix patterns. Nonetheless, the challenge of addressing previously unseen bugs persists. Recent breakthroughs in deep learning have opened up a new direction in Automatic Program Repair.
Learning-based Automatic Program Repair leverages Neural Machine Translation models from the field of Natural Language Processing. These models are based on a classic encoder-decoder architecture. The encoder processes buggy code statements and extracts relevant information for the decoder, which then generates a patch from this information. Notable examples include CoCoNut [36], Recoder [67], and DLFix [29]. Recoder stands out for its unique approach: instead of generating an entirely new statement, it edits the original statement until a correct version is achieved [67].
Although learning-based models perform remarkably well, they still face several challenges. Learning-based approaches are trained on datasets consisting of bugs and commits that fix those bugs. However, obtaining high-quality bug-fix pairs is difficult.
AlphaRepair [63] was the first to use the general-purpose code model CodeBERT to generate patches. AlphaRepair masks the buggy lines, creating a fill-in-the-blank style task for the model to solve. Various masking methods were tested, with the template mask proving to be the most effective.
ChatRepair [64] utilizes the ChatGPT model [50], which is optimized for conversations. The conversational aspect of ChatGPT is leveraged in ChatRepair to iteratively find a patch. Additionally, the model is provided with more context through test cases and failure information derived from the test cases. This approach closely resembles how a programmer would use ChatGPT for debugging.

Semantic-Based Automatic Program Repair holds a distinct position within the field of behavioural automatic program repair, due to the utilisation of symbolic execution.
A notable approach is SemFix [45], which employs symbolic execution starting from the faulty location. Symbolic execution uses symbolic values instead of concrete values to execute the program, with the goal of generating constraints for all possible program

paths. One of these paths corresponds to the desired behaviour: passing the test case. This constraint, referred to as the repair constraint, is then checked for feasibility using a constraint solver. f the constraint is feasible, a patch is generated through program synthesis, which involves creating a program that satisfies the specified constraint.
Since symbolic execution is computationally expensive due to the exponential growth of possible paths that need to be explored, the approach does not scale well to larger programs. To address this limitation, Mechtaev et al. introduced Angelix [42]. Angelix integrates a highly optimized symbolic execution phase, enabling it to efficiently handle large-scale programs.

7.2 State Automatic Program Repair

State automatic program repair attempts to correct errors during the runtime of a program. Within the subfield of state automatic program repair, a wide range of methods has been established.
One such method would be to introduce checkpoints for the possible recovery of the latest correct state [7, 51, 52].
Alternative methods include input modification, whereby a specific input that fails is modified. A strong form of modification would be to ignore the input [3, 30, 34].
In certain cases, the correct state of a program can be represented as an invariant. In the event that the current state deviates from this invariant, an attempt is made to adapt it to the correct invariant by changing the state as minimally as possible [9, 28, 46].

7.3 Metrics Used in Automatic Program Repair Research

Repair Rate is a key metric in Automatic Program Repair, measuring the percentage of faulty subjects successfully repaired. A high repair rate indicates that the employed method is versatile and broadly applicable. This metric is frequently examined in the literature [25, 31, 35, 37, 38, 57, 63, 64].

Correctness is a significant challenge for Automatic Program Repair, as the patches generated are often merely plausible rather than truly correct. A plausible patch is one that passes all tests successfully without introducing new errors. One weakness of many automatic program repair approaches is that they tend to overfit the test cases without addressing the underlying problem. Consequently, even after a 'successful' repair by an APR approach, edge cases may remain that lead to error states. Therefore, manual inspection is required to ensure the correctness of a found patch. This means that currently, these patches may only be used to assist the debugging process. The correctness of patches is an active area of research and the subject of intensive investigation [31, 35, 37, 63, 64].

Repair Time is a crucial metric for evaluating the efficiency of an Automatic Program Repair approach, indicating the time taken by the algorithm to generate a patch. The repair times of an approach are influenced by several factors, such as analysis, precomputing and test cases execution. As program complexity increases, so does the effort required for these tasks, such as template generation in CARDUMEN 2.4.4. For more complex programs with correspondingly larger test suites, the execution of test cases becomes the limiting factor [13]. The repair time is frequently considered in research [13, 25, 35, 37], with specific focus also given to the number of test case executions [47, 57].

Readability Fry et al. [14] investigated the readability of patches. Good readability contributes to better software maintainability. For each fault, there are many possible fixes. One fix could involve changing several lines, introducing new variables, and complicating the control flow, while another fix might simply add an if-statement. Although both fixes could lead to a correct solution, the shorter and more precise fix is of higher quality in terms of readability.

Usefulness Developers are often reluctant to directly adopt machine-generated code into their projects. Therefore, Tao et al. [55] examined the usefulness of machine-generated patches as a debugging aid. The usefulness strongly depends on both the correctness and readability of the patch. While high-quality patches significantly improve debugging accuracy, low-quality patches actively hinder the debugging process [55].

Miscellaneous Furthermore, the locations where a patch is applied were investigated [39]. It was also examined whether more than one patch can be found for a single bug [39, 40]. Additionally, the costs associated with an automatically generated patch were studied [57].

Related Work Summary

In this section, we introduced the different areas that have developed within APR. We began with **Search-Based** APR, which includes the approaches implemented in FIXKIT, followed by **Pattern-Based** approaches aimed at optimizing the search space. More recently, breakthroughs in machine learning have enabled **Learning-Based** approaches, such as ChatRepair. Independent of these, there is also **Semantic-Based** APR, which leverages symbolic execution, constraint solving, and program synthesis. Additionally, we examined the key **metrics** used to evaluate APR effectiveness.

8 Conclusion

In this study, we have implemented and evaluated established Automatic Program Repair approaches in Python using the student assignment benchmark. Our results show that FIXKIT achieves an overall repair rate of 21.69%, demonstrating that APR techniques can be successfully applied to Python. Of the approaches evaluated, PYGENPROG consistently identified the highest number of patches.
However, our results show that each method has unique strengths, as certain issues were repaired exclusively by certain approaches. This highlights the importance of using a diverse collection of APR strategies rather than relying on a single method.
Interestingly, the expected advantages of genetic programming in APR were not as significant as originally hypothesised. Over 50% of runs showed no change in maximum fitness over 10 generations, and over 80% showed either no improvement or only a single step increase in fitness.
These findings highlight both the potential and limitations of current APR techniques. While our results indicate that APR can automate debugging to some extent, further improvements - particularly an optimization of the search space - are needed for wider applicability in real-world scenarios.

8.1 Future Work

- Investigate whether improvements to the test suite yield different results.
- Investigate the generalizability of our results by running FIXKIT on real-world software projects.
- Improve FIXKIT's implementation, as the evaluation has revealed several critical unresolved bugs.
- Enhance the test suites to improve patch quality, ensuring the generation of correct rather than merely plausible patches.
- Extend FIXKIT with pattern-based and learning-based approaches. Research indicates that incorporating additional context leads to better results. Initially, pattern-based techniques should be explored, followed by learning-based approaches, which currently appear to be the most promising.

References

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[2] M. T. Ahvanooey, Q. Li, M. Wu, and S. Wang. A survey of genetic programming and its applications. *KSII Transactions on Internet and Information Systems (TIIS)*, 13(4):1765–1794, 2019.

[3] P. E. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *Ieee transactions on computers*, 37(4):418–425, 1988.

[4] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd international conference on software engineering*, pages 1–10, 2011.

[5] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[6] H. Cao, D. Han, F. Liu, T. Liao, C. Zhao, and J. Shi. Code similarity and location-awareness automatic program repair. *Applied Sciences*, 13(14):8519, 2023.

[7] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezze. Automatic recovery from runtime failures. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 782–791. IEEE, 2013.

[8] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 65–74. IEEE, 2010.

[9] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. *Acm sigplan notices*, 38(11):78–95, 2003.

[10] M. Eberlein. Explaining and debugging pathological program behavior. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1795–1799, Singapore Singapore, Nov. 2022.

[11] M. Eberlein, M. Smytzek, D. Steinhoefel, L. Grunske, and A. Zeller. Semantic debugging. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 438–449, 2023.

[12] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324, 2014.

[13] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954, 2009.

[14] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 177–187, 2012.

[15] A. S. George. When trust fails: Examining systemic risk in the digital economy from the 2024 crowdstrike outage. *Partners Universal Multidisciplinary Research Journal*, 1(2):134–152, 2024.

[16] S. Greenland, S. J. Senn, K. J. Rothman, J. B. Carlin, C. Poole, S. N. Goodman, and D. G. Altman. Statistical tests, p values, confidence intervals, and power: a guide to misinterpretations. *European journal of epidemiology*, 31(4):337–350, 2016.

[17] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press, 1992.

[18] J. H. Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

[19] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[20] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477, 2002.

[21] A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller. When does my program do this? learning circumstances of software behavior. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Virtual Event USA, Nov. 2020.

[22] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *2013 35th international conference on software engineering (ICSE)*, pages 802–811. IEEE, 2013.

[23] Á. Kiss, R. Hodován, and T. Gyimóthy. Hddr: a recursive variant of the hierarchical delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 16–22, 2018.

[24] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25:1980–2024, 2020.

[25] X. B. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 213–224. IEEE, 2016.

[26] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 dollar each. In *2012 34th international conference on software engineering (ICSE)*, pages 3–13. IEEE, 2012.

[27] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[28] C. Lewis and J. Whitehead. Runtime repair of software faults using event-driven monitoring. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 275–280, 2010.

[29] Y. Li, S. Wang, and T. N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, pages 602–614, 2020.

[30] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 213–222, 2005.

[31] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 31–42, 2019.

[32] X. Liu and H. Zhong. Mining stackoverflow for program repair. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*, pages 118–129. IEEE, 2018.

[33] F. Long, P. Amidon, and M. Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 727–739, 2017.

[34] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 80–90. IEEE, 2012.

[35] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178, 2015.

[36] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

[37] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22:1936–1964, 2017.

[38] M. Martinez and M. Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 441–444, 2016.

[39] M. Martinez and M. Monperrus. Open-ended exploration of the program repair search space with mined templates: the next 8935 patches for defects4j. *arXiv preprint arXiv:1712.03854*, 2017.

[40] M. Martinez and M. Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *Search-Based Software Engineering: 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings 10*, pages 65–86. Springer, 2018.

[41] M. Martinez and M. Monperrus. Astor: Exploring the design space of generate-and-validate program repair beyond genprog. *Journal of Systems and Software*, 151:65–80, 2019.

[42] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701, 2016.

[43] A. d. S. Meyer, A. A. F. Garcia, A. P. d. Souza, and C. L. d. Souza Jr. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). *Genetics and Molecular Biology*, 27:83–91, 2004.

[44] G. Misherghi and Z. Su. Hdd: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151, 2006.

[45] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[46] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102, 2009.

[47] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th international conference on software engineering*, pages 254–265, 2014.

[48] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of*

the 2015 International Symposium on Software Testing and Analysis, pages 24–36, 2015.

[49] R. Rolim, G. Soares, R. Gheyi, T. Barik, and L. D'Antoni. Learning quick fixes from code repositories. *arXiv preprint arXiv:1803.03806*, 2018.

[50] J. Schulman, B. Zoph, J. Hilton, C. Kim, J. Menick, J. Weng, J. F. Ceron Uribe, L. Fedus, L. Metz, M. Pokorny, R. G. Lopes, S. Zhao, A. Vijayvergiya, E. Sigler, A. Perelman, C. Voss, M. Heaton, J. Parish, D. Cummings, R. Nayak, V. Balcom, D. Schnurr, T. Kaftan, C. Hallacy, N. Turley, N. Deutsch, V. Goel, J. Ward, A. Konstantinidis, W. Zaremba, L. Ouyang, L. Bogdonoff, J. Gross, D. Medina, S. Yoo, T. Lee, R. Lowe, D. Mossing, J. Huizinga, R. Jiang, C. Wainwright, D. Almeida, S. Lin, M. Zhang, K. Xiao, K. Slama, S. Bills, A. Gray, J. Leike, J. Pachocki, P. Tillet, S. Jain, G. Brockman, and N. Ryder. Chatgpt: Optimizing language models for dialogue., 2022.

[51] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: automatic software self-healing using rescue points. *ACM SIGARCH Computer Architecture News*, 37(1):37–48, 2009.

[52] A. Smirnov and T.-c. Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *NDSS*, 2005.

[53] M. Smytzek and A. Zeller. Sflkit: A workbench for statistical fault localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1701–1705, 2022.

[54] S. H. Tan and A. Roychoudhury. relifix: Automated repair of software regressions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 471–482. IEEE, 2015.

[55] Y. Tao, J. Kim, S. Kim, and C. Xu. Automatically generated patches as debugging aids: a human study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 64–74, 2014.

[56] R. L. Wasserstein and N. A. Lazar. The asa statement on p-values: context, process, and purpose, 2016.

[57] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 356–366. IEEE, 2013.

[58] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374. IEEE, 2009.

[59] M. Weiser. Program slicing. *IEEE Transactions on software engineering*, SE-10(4):352–357, 1984.

[60] M. D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* University of Michigan, 1979.

[61] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th international conference on software engineering*, pages 1–11, 2018.

[62] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[63] C. S. Xia and L. Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971, 2022.

[64] C. S. Xia and L. Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 819–831, 2024.

[65] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. *RN*, 14(14):14, 2014.

[66] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on software engineering*, 28(2):183–200, 2002.

[67] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 341–353, 2021.

Appendix

Seed	PyGenProg	PyCardumen	PyKali	PyMutRepair
0	120	105	94	88
1343	120	89	81	92
2419	120	63	88	89
3798	117	99	90	86
5637	122	95	91	87
6056	115	89	90	85
6841	108	99	94	87
6924	120	81	98	89
7770	128	92	84	87
8013	101	95	84	87

Table 11: Overview of the patches found by each approach across all seeds

Candidates	PyKali	PyMutRepair
average q1	28.2	35.0
median q1	25.0	31.0
average q2	89.8	112.0
median q2	97.0	121.0
average q3	33.8	42.0
median q3	37.0	46.0
average q4	62.6	78.0
median q4	57.0	71.0
average all	53.6	66.75
median all	49.0	61.0

Table 12: Overview of created candidates by PYKALI and PYMUTREPAIR.

Statistical Hypothesis Tests of the Repairs

Kolmogorov-Smirnov Test The Kolmogorov-Smirnov test is a nonparametric statistical test that compares the cumulative distribution functions of two datasets to determine if they come from the same distribution.

Null hypothesis H_0 The distribution of the sample is identical to the normal distribution.

Alternative hypothesis H_1 The distribution of the sample differs from the normal distribution.

Approach	D-Statistics	P-Value	Result
PyGenProg	0.25	0.49	No Rejection of H_0
PyCardumen	0.24	0.52	No Rejection of H_0
PyKali	0.14	0.96	No Rejection of H_0
PyMutRepair	0.24	0.53	No Rejection of H_0

Table 13: Results of the Kolmogorov-Smirnov test

Shapiro-Wilk Test The Shapiro-Wilk test is a statistical test that assesses whether a given dataset follows a normal distribution.

Null hypothesis H_0 The population of the sample is normally distributed.

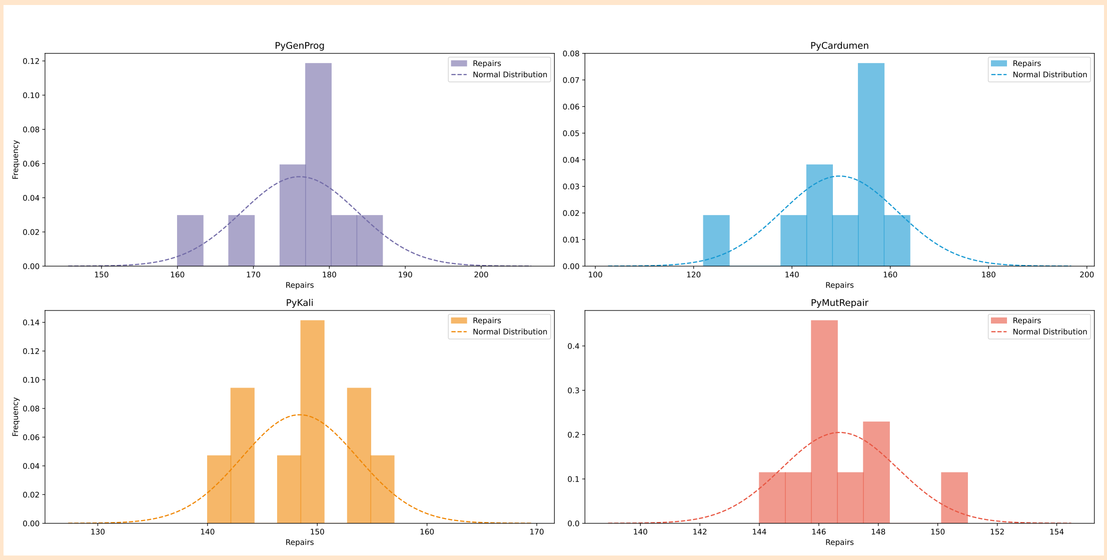
Alternative hypothesis H_1 The population of the sample is not normally distributed.

Approach	W-Statistics	P-Value	Result
PyGenProg	0.88	0.15	No Rejection of H_0
PyCardumen	0.87	0.11	No Rejection of H_0
PyKali	0.96	0.15	No Rejection of H_0
PyMutRepair	0.90	0.24	No Rejection of H_0

Table 14: Results of the Shapiro-Wilk Test

Interpretation The tests do not reject the null hypothesis. They indicate that the data are normally distributed. As a further check, we look at the data graphically.

Continuation on the next page.



Interpretation We conclude that the data most likely do not follow a normal distribution, with the exception of the data from PYKALI. Therefore, we need to continue with a non-parametric test.

Kruskal-Wallis Test The Kruskal-Wallis test is a non-parametric test that analyses whether there is a significant difference between the medians of the samples.

> **Nullhypothese** H_0 There is no difference between the medians of the groups.
>
> **Alternativhypothese** H_1 There is at least one difference between the medians of the groups.

Approach	H-Statistics	P-Value	Result
All Approaches	24.02	2.46×10^{-5}	Rejection of H_0

Table 15: Results of the Kruskal-Wallis Test

Interpretation The null hypothesis can be rejected. The mean values of at least one group differ significantly from the others. Next, a posthoc-dunn test can be performed to pairwise compare the approaches.

Posthoc Dunn Test The Dunn test is a post-hoc test used after a Kruskal-Wallis test to perform pairwise comparisons between the groups and determine which groups differ significantly from each other.

Continuation on the next page.

Null hypothesis H_0 There is no difference between the medians of the two groups compared

Alternative hypothesis H_1 There is a significant difference between the medians of the two groups compared

Group Comparison	P-Value	Result	Mean Difference
PyGenProg vs PyCardumen	0.0139	Rejection of H_0	26.4
PyGenProg vs PyKali	0.0008	Rejection of H_0	27.7
PyGenProg vs PyMutRepair	0.00003	Rejection of H_0	29.4
PyCardumen vs PyKali	1.0000	No Rejection of H_0	1.3
PyCardumen vs PyMutRepair	0.7805	No Rejection of H_0	3.0
PyKali vs PyMutRepair	1.0000	No Rejection of H_0	1.7

Table 16: Results of pairwise comparisons of the post-hoc Dunn test

Interpretation All approaches differ significantly from PYGENPROG. The difference in the mean shows that PYGENPROG finds significantly more repairs than the other approaches.

All the test results have been presented, allowing the reader to form their own interpretation of the values.

Statistical Hypothesis Tests of the Runtime

Kolmogorov-Smirnov Test The Kolmogorov-Smirnov test is a nonparametric statistical test that compares the cumulative distribution functions of two datasets to determine if they come from the same distribution.

Null hypothesis H_0 The distribution of the sample is identical to the normal distribution.

Alternative hypothesis H_1 The distribution of the sample differs from the normal distribution.

Approach	D-Statistics	P-Value	Result
PyGenProg	0.19	0	Rejection of H_0
PyCardumen	0.13	8.6×10^{-266}	Rejection of H_0
PyKali	0.18	0	Rejection of H_0
PyMutRepair	0.23	0	Rejection of H_0

Table 17: Results of the Kolmogorov-Smirnov test

Shapiro-Wilk Test The Shapiro-Wilk test is a statistical test that assesses whether a given dataset follows a normal distribution.

Null hypothesis H_0 The population of the sample is normally distributed.

Alternative hypothesis H_1 The population of the sample is not normally distributed.

Approach	W-Statistics	P-Value	Result
PyGenProg	0.70	9.1×10^{-99}	Rejection of H_0
PyCardumen	0.89	5.9×10^{-75}	Rejection of H_0
PyKali	0.68	$5.3e \times 10^{-101}$	Rejection of H_0
PyMutRepair	0.55	$2.4e \times 10^{-109}$	Rejection of H_0

Table 18: Results of the Shapiro-Wilk Test

Interpretation The data are not normally distributed and a non-parametric test is required to check for differences.

Continuation on the next page.

Kruskal-Wallis Test The Kruskal-Wallis test is a non-parametric test that analyses whether there is a significant difference between the medians of the samples.

Nullhypothese H_0 There is no difference between the medians of the groups.

Alternativhypothese H_1 There is at least one difference between the medians of the groups.

Approach	H-Statistics	P-Value	Result
All Approaches	37,257.37	0	Rejection of H_0

Table 19: Results of the Kruskal-Wallis Test

Interpretation The null hypothesis can be rejected. The mean values of at least one group differ significantly from the others. Next, a posthoc-dunn test can be performed to pairwise compare the approaches.

Posthoc Dunn Test The Dunn test is a post-hoc test used after a Kruskal-Wallis test to perform pairwise comparisons between the groups and determine which groups differ significantly from each other.

Null hypothesis H_0 There is no difference between the medians of the two groups compared

Alternative hypothesis H_1 There is a significant difference between the medians of the two groups compared

Group Comparison	P-Value	Result
PyCardumen vs PyGenProg	0	Rejection of H_0
PyCardumen vs PyKali	0	Rejection of H_0
PyCardumen vs PyMutRepair	0	Rejection of H_0
PyGenProg vs PyKali	0	Rejection of H_0
PyGenProg vs PyMutRepair	1.5×10^{-270}	Rejection of H_0
PyKali vs PyMutRepair	1.1×10^{-104}	Rejection of H_0

Table 20: Results of pairwise comparisons of the post-hoc Dunn test

Interpretation All groups differ significantly in their runtime.

All the test results have been presented, allowing the reader to form their own interpretation of the values.

Statistical Hypothesis Tests of the Time to Fix

Kolmogorov-Smirnov Test The Kolmogorov-Smirnov test is a nonparametric statistical test that compares the cumulative distribution functions of two datasets to determine if they come from the same distribution.

Null hypothesis H_0 The distribution of the sample is identical to the normal distribution.

Alternative hypothesis H_1 The distribution of the sample differs from the normal distribution.

Approach	D-Statistics	P-Value	Result
PyGenProg	0.28	9.6×10^{-88}	Rejection of H_0
PyCardumen	0.23	2.7×10^{-41}	Rejection of H_0
PyKali	0.31	3.7×10^{-78}	Rejection of H_0
PyMutRepair	0.21	9.5×10^{-36}	Rejection of H_0

Table 21: Results of the Kolmogorov-Smirnov test

Shapiro-Wilk Test The Shapiro-Wilk test is a statistical test that assesses whether a given dataset follows a normal distribution.

Null hypothesis H_0 The population of the sample is normally distributed.

Alternative hypothesis H_1 The population of the sample is not normally distributed.

Approach	W-Statistics	P-Value	Result
PyGenProg	0.50	3.7×10^{-49}	Rejection of H_0
PyCardumen	0.63	4.2×10^{-40}	Rejection of H_0
PyKali	0.38	1.1×10^{-47}	Rejection of H_0
PyMutRepair	0.65	4.1×10^{-39}	Rejection of H_0

Table 22: Results of the Shapiro-Wilk Test

Interpretation The data are not normally distributed and a non-parametric test is required to check for differences.

Continuation on the next page.

Kruskal-Wallis Test The Kruskal-Wallis test is a non-parametric test that analyses whether there is a significant difference between the medians of the samples.

> **Nullhypothese** H_0 There is no difference between the medians of the groups.
>
> **Alternativhypothese** H_1 There is at least one difference between the medians of the groups.

Approach	H-Statistics	P-Value	Result
All Approaches	1,501.69	0	Rejection of H_0

Table 23: Results of the Kruskal-Wallis Test

Interpretation The null hypothesis can be rejected. The mean values of at least one group differ significantly from the others. Next, a posthoc-dunn test can be performed to pairwise compare the approaches.

Posthoc Dunn Test The Dunn test is a post-hoc test used after a Kruskal-Wallis test to perform pairwise comparisons between the groups and determine which groups differ significantly from each other.

> **Null hypothesis** H_0 There is no difference between the medians of the two groups compared
>
> **Alternative hypothesis** H_1 There is a significant difference between the medians of the two groups compared

Group Comparison	P-Value	Result
PyCardumen vs PyGenProg	4.91×10^{-179}	Rejection of H_0
PyCardumen vs PyKali	7.34×10^{-208}	Rejection of H_0
PyCardumen vs PyMutRepair	8.99×10^{-6}	Rejection of H_0
PyGenProg vs PyKali	1.36×10^{-4}	Rejection of H_0
PyGenProg vs PyMutRepair	3.28×10^{-119}	Rejection of H_0
PyKali vs PyMutRepair	1.76×10^{-146}	Rejection of H_0

Table 24: Results of pairwise comparisons of the post-hoc Dunn test

Interpretation All groups differ significantly in their time to fix.

All the test results have been presented, allowing the reader to form their own interpretation of the values.

Patch	**Question**	**Subject Number**	**Seed**
Patch 1	1	1	0
Patch 2	1	2	0
Patch 3	1	3	0
Patch 4	1	4	0
Patch 5	1	5	0
Patch 6	1	6	0
Patch 7	1	7	0
Patch 8	1	8	0
Patch 9	1	9	0
Patch 10	1	10	0

Table 25: Translation of patch to run

No Patch	**Question**	**Subject Number**	**Seed**
No Patch 1	1	22	0
No Patch 2	1	26	0
No Patch 3	1	27	0
No Patch 4	1	31	0
No Patch 5	1	54	0
No Patch 6	1	59	0
No Patch 7	1	61	0
No Patch 8	1	68	0
No Patch 9	1	69	0
No Patch 10	1	72	0

Table 26: Translation of no patch to run

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den February 10, 2025 [illegible]