Seeding the Pynguin: Improving Test Coverage through String Constraint-Guided Initialization *

Samuel Brinkmann[1], Trung Kien Hoang[1], Levin Schulz[1], and Sharui Yang[1]

Humboldt University, Unter den Linden 6, 10117 Berlin, Germany
{samuel.brinkmann, trung.kien.hoang, levin.schulz, sharui.yang}@student.hu-berlin.de

Abstract. While genetic algorithms (GAs) lead test generation techniques for code coverage in Python, they can be less effective for functions with string-based conditions, due to inefficient random initialization. This paper introduces a string constraint-guided seeding strategy that improves GA initialization in Pynguin, a Python-based test generation tool. By traversing a function's abstract syntax tree (AST) to extract string literals, the strategy seeds the initial population with contextually relevant inputs, fully in-memory and with minimal overhead. Empirical evaluation shows improved statement coverage for targeted internal tests. However, results on external benchmarks indicate limited generalizability in the current form. These findings highlight both the potential and the current constraints of guided seeding in dynamic languages. Looking ahead, broader input type support and static data flow analysis may enhance applicability. Nevertheless, this work contributes a practical step toward more effective, constraint-aware automated testing in Python and advances the goal of improving software quality with minimal manual effort.

Keywords: test generation · genetic algorithm · string constraints

1 Introduction

Recent studies show that test automation maturity has benefits for both product quality and the release cycle frequency [16]. However, writing test cases can introduce a heavy workload. This burden can be alleviated with automated test generation tools, which help to achieve high statement coverage of the code with low effort [7]. Furthermore, studies show that early testing in the project lifecycle can also improve product quality [15]. This aligns with the low-effort nature of automated test generation, making early testing more feasible. As a result, automated test generation emerges as a desirable strategy for efficiently improving software quality.

One tool to possibly achieve all this in the programming language Python is Pynguin [8]. Pynguin's test generation is driven by a genetic algorithm (GA),

* Supported by Institut für Informatik - Software Engineering.

which uses a randomly generated initial population or a seed given as a file. To keep the effort for automated testing low, the seeds given as a file cannot be generated manually, as this would require similar effort and reasoning as non-automated test generation—precisely what we aim to avoid. Instead, they would need to be generated automatically, which leads to a disjointed integration, as the data must first be written to disk and then read back in. Thus, random initialization remains the most practical low-effort option. However, Pynguin demonstrates significant inefficiencies when using the random initialization to test functions with simple string-based conditions. The algorithm relies heavily on random mutations to discover input values that satisfy trivial constraints, resulting in excessive computational overhead and slow convergence.

Consider a basic substring-checking function like shown in Listing 1.1. Even after ten seconds of running the GA, Pynguin is not able to generate a test case that covers line 3.

```
1 def in_func(param: str) -> bool:
2     if "test123" in param:
3         return True
4     return False
```

Listing 1.1. Example function featuring an "`in`" expression.

This paper presents an extension of Pynguin 0.41.0 that analyzes a function's abstract syntax tree (AST) to enable string constraint-guided seeding. The approach is designed to be as time-efficient as random GA initialization and passes the seeds in-memory to ensure seamless integration. The goal is to improve Pynguin's statement coverage further in order to fully realize the benefits of automated test generation.

Contributions. The main contributions of this work are:

- An extension to the Pynguin tool that enables AST-based extraction of string literals for guided seeding in the initial population of the genetic algorithm.
- A fully in-memory integration of the seeding mechanism, avoiding file-based seed input and preserving Pynguin's low-effort workflow.
- An adaptation of the constants extraction seeding strategy from Java-based SBST to the dynamic typing model of Python.
- An empirical evaluation demonstrating improved statement coverage on functions with string-based conditions, compared to Pynguin's default random initialization.

2 State of the Art

2.1 Automated Test Generation and Pynguin

Pynguin is a well-established tool for automatic unit test generation in Python. It led the Search-Based Software Testing (SBST) Tool Competition 2024 in terms

of statements and branches covered [3]. It employs a modular architecture and supports both random and search-based techniques, such as genetic algorithms [4,5]. A recent empirical study demonstrates that advanced GA strategies like DynaMOSA outperform purely random approaches in terms of code coverage and test effectiveness [7]. However, the dynamic nature of Python—including the absence of static type information—presents ongoing challenges for search-based testing tools like Pynguin. Furthermore, string-based conditions, such as substring checks, remain notably difficult for the genetic algorithm to handle effectively.

2.2 Seeding Strategies in Search-Based Test Generation

Prior research in SBST has established that intelligent seeding can significantly improve the efficiency of test generation. In particular, a large-scale evaluation of seeding strategies in Java projects showed that, among other approaches, using constants extracted from source code leads to statistically significant improvements in coverage and mutation scores [9]. Despite these promising results, such techniques have not yet been thoroughly explored in the context of Python's dynamic type system and runtime behavior.

2.3 Static, Symbolic, and LLM-Based Test Generation

Beyond Pynguin, several other tools have been proposed to improve automated test generation for Python. Klara, introduced as part of the SBST Tool Competition 2024, statically analyzes the abstract syntax tree (AST) to extract test inputs [2]. However, it does not incorporate search-based feedback or dynamic optimization.

On the other hand, UTBotPython combines symbolic execution with fuzzing to improve input diversity and constraint satisfaction, but requires substantial infrastructure and computational resources [13].

In parallel, approaches based on large language models (LLMs), such as PolyTest, have shown improvements over Pynguin in terms of statement coverage, branch coverage, and mutation score [6]. However, these methods rely on hosting resource-intensive models such as LLaMA 3–70B or GPT-4o, and often suffer from issues with reproducibility and explainability.

2.4 String Constraint-Guided Initialization

In contrast to prior work, our extension to Pynguin is unique in that it:

- Combines AST-based static analysis with dynamic GA-based optimization;
- Operates fully in-memory, avoiding file-based overhead and integration complexity;
- Implements effective seeding strategies proven successful in Java-based SBST;
- Remains lightweight, solver-free, and model-free, preserving usability, reproducibility, and performance.

This positions our approach as a practical enhancement to existing GA-based test generation for Python, addressing known inefficiencies in handling string constraints and filling a gap left by purely static, symbolic, and LLM-based alternatives.

3 Background

Before discussing our solution design and implementation, we establish common ground on Genetic Algorithms and Abstract Syntax Trees, which are central to understanding the underlying mechanisms of our approach.

3.1 Genetic Algorithms

Genetic Algorithms (GAs) are population-based search heuristics inspired by natural evolution. In automated test generation, each individual represents a candidate test case, and the goal is to evolve test cases that maximize a fitness function, such as statement coverage.

As shown in Figure 1, the process begins with an initial population, which can be generated randomly or guided through seeding. Next, this population is evaluated, and the candidates with the lowest fitness values are removed. The remaining test cases function as the parents for the next generation.

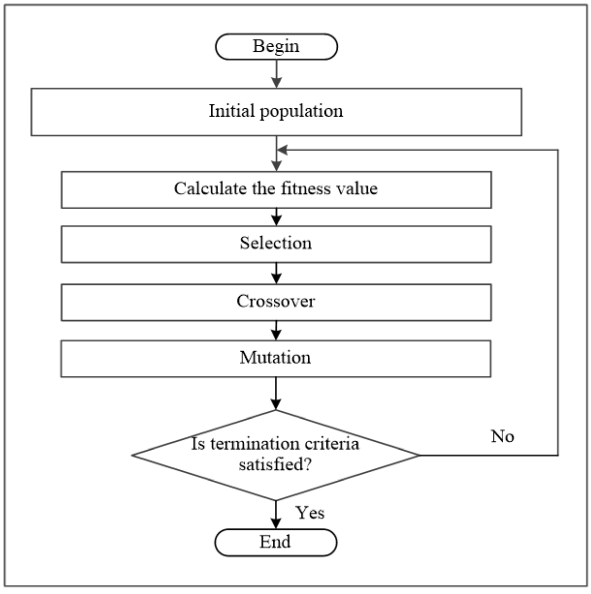

Fig. 1. General workflow of a genetic algorithm, including initialization, selection, crossover, mutation, and fitness evaluation. Reproduced from [1].

New individuals are created through **crossover**, which combines parts of two parents, and **mutation**, which introduces random changes to a parent to maintain diversity within the population. Afterwards, the GA checks if the specified termination criteria, such as time budget or a certain coverage level, are satisfied. If not, the cycle of evaluation and reproduction starts again.

The quality of the initial population strongly influences convergence speed. Poor initialization can lead to inefficient searches, particularly in cases where specific constraints—such as string comparisons—must be satisfied by chance.

3.2 Abstract Syntax Trees

An Abstract Syntax Tree (AST) is a tree-based representation of the syntactic structure of source code. In an AST, each node corresponds to a syntactic construct, such as a function definition, a conditional statement, or an expression. ASTs abstract away specific syntax elements like parentheses or formatting, thereby enabling structural analysis of code. In our case, Pynguin creates an AST for each function under testing that we will use in our approach.

4 Solution Design

Let us now look at the solution design of our approach. In Figure 2, we can see the original workflow of Pynguin before our extension. Pynguin has two flags for seeding—`initial_population_seeding` for activating or deactivating the seeding and `initial_population_data` for passing the file path of the seeds.

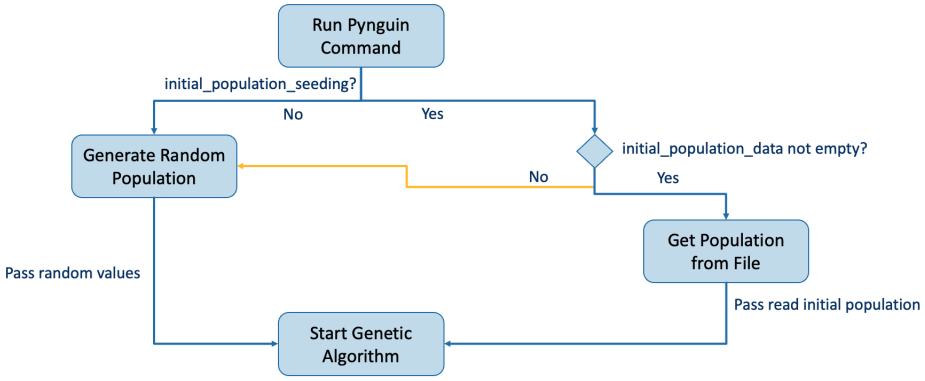

Fig. 2. Pynguin's original population initialization workflow (before AST-based seeding extension).

In the original flow, Pynguin uses random initialization for the GA if seeding is activated, but the seed file path is left empty. This behavior stems from the use of two separate configuration flags—one for enabling seeding and one for specifying the file path. While this is a practical solution within the original design, it is unintuitive from a user perspective: seeding appears to be enabled, yet no actual seeding occurs unless both flags are correctly set.

Our approach resolves this inconsistency in a clean and integrated way by interpreting the absence of a seed file as a fallback trigger for in-memory AST-based seeding (see Figure 3). Furthermore, we introduced an additional configuration flag, `initial_population_strategy`, to control the fallback seeding strategy, which defaults to the AST-based approach. This design allows for easy integration of alternative strategies by leveraging the interface introduced in this work.

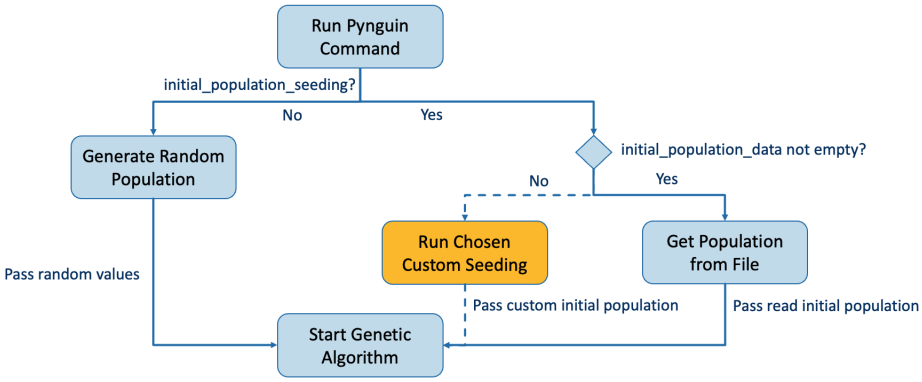

Fig. 3. Modified initialization workflow in Pynguin with AST-based in-memory seeding. The genetic algorithm uses string literals extracted from the AST when no external seed file is provided.

Next, we discuss the design of the seeder in more detail. It consists of three main components: the **node traverser**, which traverses the AST; the **node handler**, which selects the appropriate extractor for each node; and the **node extractors**, which extract relevant information from the nodes (see Figure 4). The input to the seeding strategy consists of the function's AST and its input parameter names. Functions that do not consist only of string-typed input parameters are excluded from the seeding process. This filtering is based on the type hints provided in the function signature; functions without type hints are also excluded. In those cases, Pynguin falls back to a random initialization. The output of the seeder is a list of test cases for the function, which are passed to the genetic algorithm.

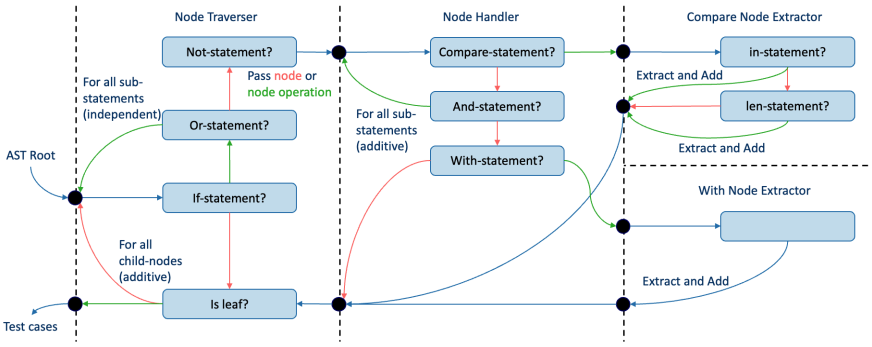

Fig. 4. Workflow of the AST-based string constraint extractor used for guided seeding. The process begins at the AST root and traverses the tree through various control-flow constructs (e.g., `if`, `or`, `not`, `with`, `compare` statements). Nodes of interest are passed to specialized extractors (e.g., `in`, `len`, `with`), which identify string-related constraints. Relevant literals are collected and later used to seed the initial population of the genetic algorithm. Arrows indicate traversal paths: blue for default flow, green for "yes" decisions, and red for "no" decisions. The `with` extractor handles string prefix and suffix checks, such as `startswith()` and `endswith()`. Traversals marked as *additive* contribute extracted information that is passed along and aggregated across subsequent child or sub-statement nodes, while *independent* nodes are handled separately, with their extracted values used in isolation.

We do not discuss the internal components in greater detail, as the presented flow is intended to convey the core logic relevant to this work, rather than exhaustively document implementation specifics. The exact implementation is available in our project's GitHub repository [10]. To aid understanding, we revisit the workflow with an illustrative example in the next Section.

It is worth noting that the current design has limitations. For instance, due to the separation of logic between the node handler (handling "`and`") and the node traverser (handling "`or`"), expressions such as "`(... or ...) and ...`" are not processed correctly—specifically, the left-hand side of the "`and`" is skipped. Addressing such issues is left for future improvements.

5 Implementation

Having established the solution design, we now turn to its implementation. The source code for this extension is available in the GitHub repository `penguin_and_seeding` [10]. Since the setup and all code changes are documented in detail in the repository, we omit a line-by-line explanation and instead summarize the key modifications made to extend Pynguin.

For modifying the Pynguin workflow (see Figure 3), changes were made to the following files:

- src/pynguin/analyses/seeding.py,
- src/pynguin/ga/generationalgorithmfactory.py,
- src/pynguin/configuration.py,
- src/pynguin/generator.py.

In addition, a new folder named `custom_seeding` was added to the Pynguin package to contain the implementations of the seeding strategies (see initial changes in [11]).

The implementation of the strategy shown in Figure 4 can be found in the file `tree_traverse_strategy.py` located in the folder `pynguin_0_41_0/src/pynguin/custom_seeding/strategy/`.

Seed Generation Process

To better understand the seed generation process, we illustrate it using an example. Listing 1.2 shows a function with two string-typed input parameters, two nested `if`-statements, and one standalone `if`-statement. Additionally, it includes logical operators such as "`or`" and "`and`" within the `if`-conditions, as well as various string operations. This function serves as a suitable example to demonstrate different aspects of the implementation.

```
1 def example(par1: str, par2: str) -> bool:
2     if len(par1) < len(par2) or "testcase" in par2:
3         if not "test" in par2:
4             return True
5     if len(par2) > 7 and par2.startswith("start"):
6         return True
7     return False
```

Listing 1.2. Example function used to illustrate the implementation process.

In Figure 5, we show the AST at the point where it encounters the first condition (`len(par1) < len(par2)`, line 2) during traversal. When comparing the lengths of two input parameters without any prior string constraints, we generate two new test cases. The first is a minimal case, where the parameters have length one or zero, depending on the comparison operator. The second is an "extreme" case, where the difference in parameter lengths is just sufficient to satisfy the condition, but additional symbols are appended (e.g., parameters of lengths 9 and 10). The rationale is that, even if we extend the shorter parameter—`par1` in this instance—due to substring constraints, a crossover between the two test cases can yield a valid input that satisfies both the length and substring conditions. The goal is not to generate test cases that already cover all statements, but to provide a foundation that can, with a few iterations of the GA, evolve into such cases. This approach reduces the overhead of seed generation while maintaining high performance.

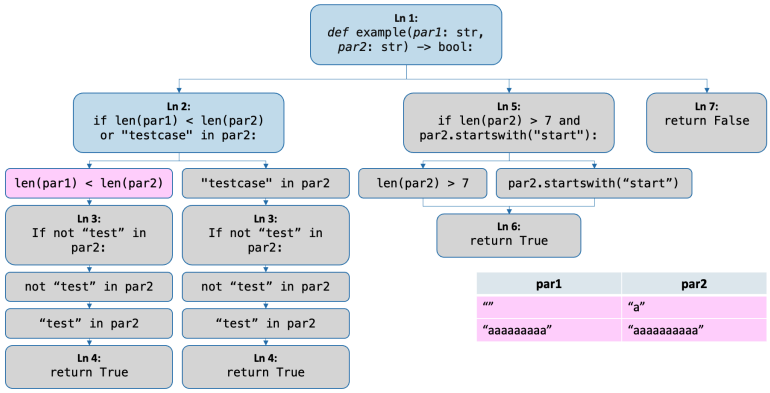

Fig. 5. Step-by-step visualization of the seed generation process based on AST traversal. The function is decomposed into logical branches, and relevant string-related conditions are extracted to construct targeted test cases. Step 1: AST traversal for the node involving the condition `len(par1) < len(par2)`.

As we see in Figure 6, "`not <string> in <parameter>`" statements are handled like "`<string> in <parameter>`" statements, as this represents the more restrictive condition. Furthermore, as mentioned earlier, the test case does not need to satisfy the condition initially; a single mutation of the string may suffice to satisfy the "`not`". In "`<string> in <parameter>`" cases, the substring constraint is appended to all test cases along this branch of the AST. Again, had it been `par1` instead of `par2`, the individual test cases would not have satisfied the previous length constraint; however, a crossover between them would have.

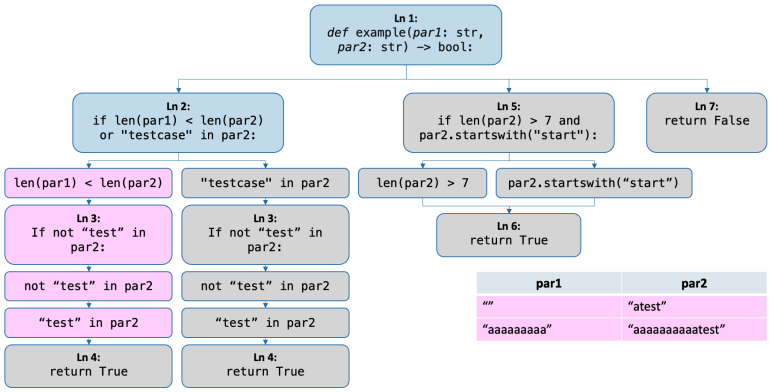

Fig. 6. Step 2: AST traversal for the node involving the condition `not "test" in par2`.

It should be noted that the current implementation is not yet fully refined and should be viewed as a proof of concept. For example, in cases where an "`endswith`" constraint is followed by an "`in`" check within a nested conditional structure, the current logic appends the constraint for both, potentially overwriting the satisfaction of the initial "`endswith`" condition. Such logical inconsistencies can be addressed in future improvements as the extension matures.

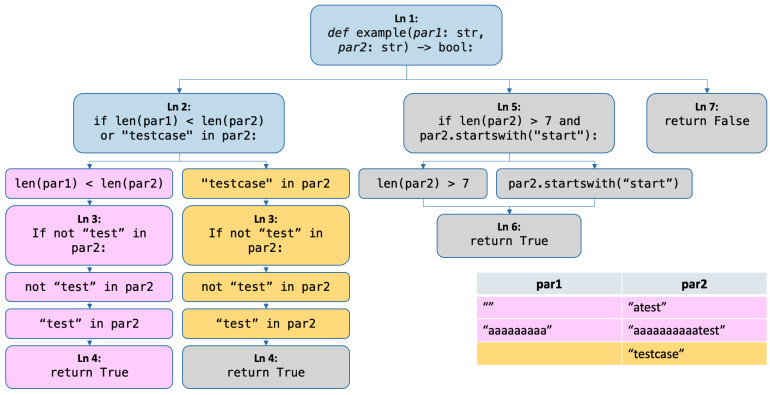

Fig. 7. Step 3: AST traversal along the yellow branch, involving a second "`in`" expression that is already satisfied.

In the case of two "`in`" expressions where the second is satisfied as a consequence of the first, the test cases are not extended, in order to avoid unnecessarily long test inputs (see Figure 7). The value for `par1` is left empty, as there is no associated constraint.

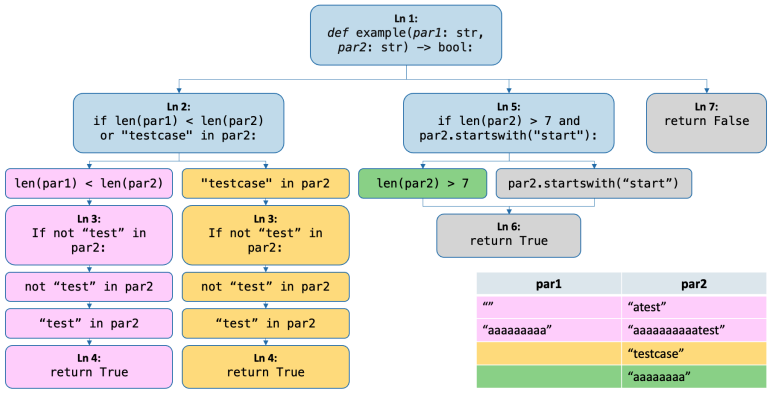

Fig. 8. Step 4: AST traversal for the node involving an "`and`" statement.

As shown in Figure 8, "`and`" statements are handled like nested `if`-statements; that is, the constraints are applied cumulatively to the test cases along this branch. In this example, the length constraint is applied first, followed by the "`startswith`" condition, which is satisfied by prepending the specified literal to all test cases in this branch. This behavior can be observed by comparing the green test case values between Figure 8 and Figure 9.

Finally, test cases in which input parameters are not assigned values—due to missing constraints—are completed with empty strings. This concludes the example seed generation process. As emphasized throughout, the goal is to generate a foundation that the GA can build upon using crossovers and mutations to improve statement coverage, rather than solving the test generation problem directly via AST analysis, as Klara does [2].

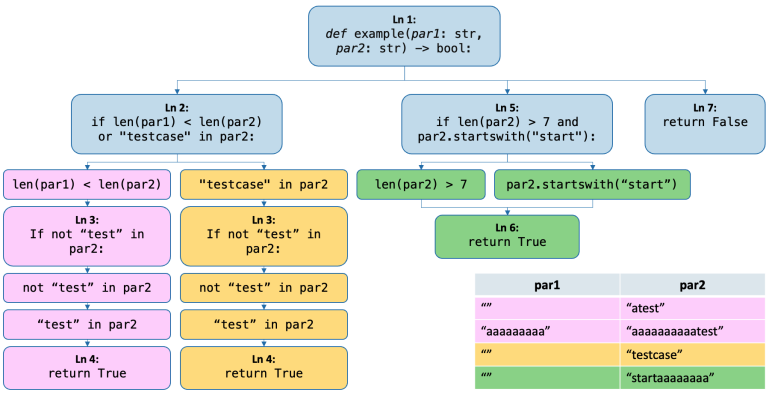

Fig. 9. Final state of the AST traversal and test cases.

6 Evaluation Design

After discussing the solution design and implementation of our string constraint-guided seeding approach, the next step is to evaluate it and compare it with Pynguin configured with random initialization. For this purpose, we conduct both internal and external tests.

The internal tests comprise 64 cases that systematically cover logical constructs such as nested conditions, `and`, `or`, etc., and edge cases involving "`in`" (35 tests), "`len`" (17 tests), "`startswith`" and "`endswith`" (6 tests), and mixed statements (6 tests). Mixed statements are combinations of the other constraint types. For the internal tests, one file corresponds to one test.

By contrast, the external tests consist of 52 files from the `TheAlgorithms` repository, specifically from the "`string`" functions folder [14]. One file corresponds to one or multiple tests. This repository was selected for two reasons. First, a practical consideration: it is part of `ManyTypes4Py`, a benchmark comprising over 5,000 repositories that use type hints—a requirement for identifying

string parameters in our approach. Moreover, unlike many of the 50+ repositories from `ManyTypes4Py` we examined, it does not require a complex setup involving localhost configurations, file I/O, or external API calls. Second, with 1,226 contributors (as of July 17, 2025), the repository reflects a diversity of coding styles. Consequently, its test cases represent real-world code and help assess whether the improvements generalize to practical scenarios, thereby supporting their relevance.

The evaluation will be conducted as follows: for both approaches—Pynguin configured with random initialization (`None`) and Pynguin with string constraint-guided seeding (`tree_traverse`)—we will run each file with five random seeds (the same for both approaches). For each file, each run will be given 10 seconds for either the genetic algorithm (GA) alone or the seeding step followed by the GA[1]. The improvement will be measured in terms of achieved statement coverage. All evaluations are conducted in an automated fashion to ensure consistency and reproducibility. The random seeds used for each run are documented in the repository as part of the experiment settings (`scripts/config.py`) [10].

To assess whether observed differences in coverage between the two approaches are statistically significant, we will apply the paired Wilcoxon signed-rank test. This non-parametric test is appropriate given the paired structure of the data, the potential for non-normality in coverage distributions, and its robustness to outliers and small sample sizes [12, see Test 18].

7 Evaluation

With the evaluation setup in place, we now turn to the analysis of the results. In Figure 10, we see the average statement coverage for the internal and external tests across the five random seeds for both approaches.

We begin by discussing the results of the internal tests. We observe an increase of more than 20% in the average statement coverage for functions involving "`in`", "`startswith`" and "`endswith`", and mixed logic. In contrast, the coverage for the "`len`" functions shows no increase or decrease, indicating that Pynguin already achieves high coverage on these cases. To support this observation statistically, we performed a paired Wilcoxon signed-rank test. As shown in Figure 11, the new approach is consistently equal to or better than the base Pynguin across all runs. Similarly, Table 1 confirms a statistically significant difference in performance between the two approaches on internal tests. The Wilcoxon statistic of 0.0 implies that in all (or nearly all) paired comparisons, the seeding approach performed better. The p-value is effectively zero, confirming that this difference is highly significant.

[1] As Pynguin only accepts integers as the time budget, the GA typically receives 9 seconds in the `tree_traverse` approach, even though the seeding step takes only 1–3 ms.

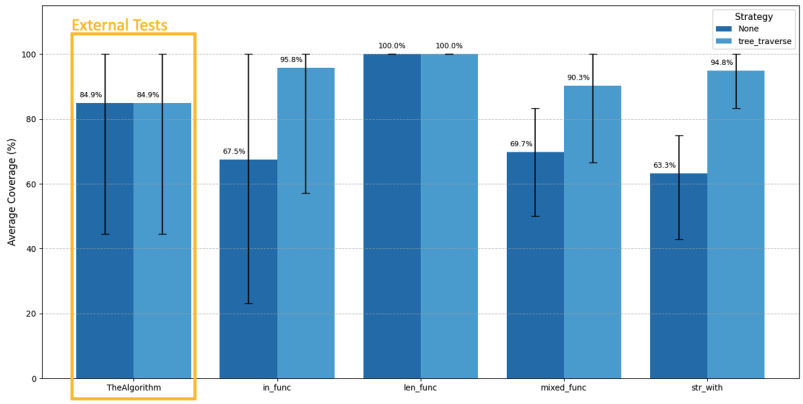

Fig. 10. Average statement coverage across five runs with different random seeds for both external and internal tests. The first two bars (under the first x-axis label) represent the external tests and are highlighted in the orange box. The remaining bars correspond to internal tests, grouped by function type (e.g., `in` expressions, `len` expressions, etc.), with each group tested across multiple systematically generated files. The y-axis shows the mean statement coverage (%), and error bars indicate the minimum and maximum values across the five runs. Results are shown for two configurations: the baseline (`None`), which corresponds to Pynguin without seeding, and `tree_traverse`, our string constraint-guided seeding approach.

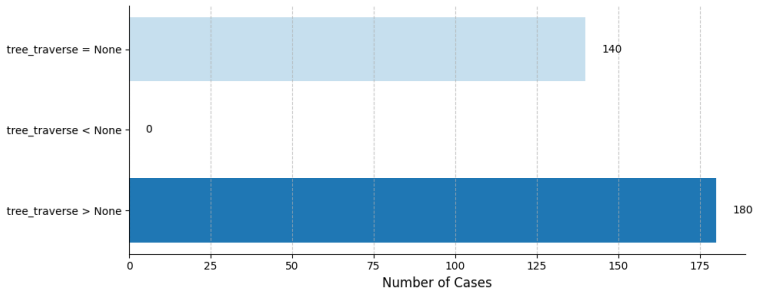

Fig. 11. Results of the paired Wilcoxon signed-rank test for **internal** test comparisons, showing the distribution of cases where `tree_traverse` performs better than, worse than, or equal to `None`. Pairings are based on matching file and seed combinations.

While partial improvements are observed in the internal tests, Figure 12 and Table 1 show that there is no significant difference between the two approaches on the external tests. A p-value of 1.0 indicates that the performance differences observed could entirely be due to chance, and the Wilcoxon statistic suggests that these differences are minimal and not consistently in one direction. A closer

examination of the external tests reveals that the seeder generated seeds for only nine functions. This is primarily because many of the functions do not have solely `str`-typed input parameters, do not contain `if`-statements, or include `if`-statements that either use custom logic, call unsupported functions, or do not involve the input parameters directly in the condition.

In the following discussion, we reflect on what these findings reveal about the strengths and limitations of the proposed seeding approach.

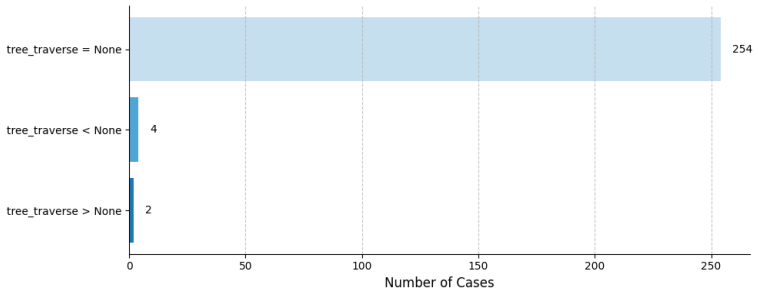

Fig. 12. Results of the paired Wilcoxon signed-rank test for **external** test comparisons, showing the distribution of cases where `tree_traverse` performs better than, worse than, or equal to `None`. Pairings are based on matching file and seed combinations.

Table 1. Wilcoxon Test Statistics and p-Values for Internal and External Tests. The Wilcoxon statistic reflects the sum of ranks of the less frequent differences in pairwise comparisons of statement coverage between Pynguin with random initialization and string constraint-guided seeding (`tree_traverse`).

	Internal Tests	External Tests
Wilcoxon Statistic	0.0	10.5
p-Value	9.2e-32	1.0

8 Discussion

Let us start by discussing the strengths of the extension. We observed a significant improvement in statement coverage for the internal tests, indicating that Pynguin is now able to cover string constraints that were previously challenging. The seeding step itself required only 1–3 ms per file. As mentioned earlier, the GA was given only 9 seconds in the extension configuration, so one could reasonably assume that with a smaller reduction in GA time due to seeding, the fact that the extension performs equally or better than baseline Pynguin would be even more pronounced.

However, there are also limitations. The internal tests do not reflect real-world appearances and relevance. On the external tests, no improvement was observed. This leads to the conclusion that, in its current form, the extension provides a modest improvement to Pynguin, but with limited applicability.

As the evaluation was limited to a single repository with small, independent modules and excluded scenarios such as localhost setups or file handling, some practical use cases may have been missed. One could hypothesize that, particularly in situations where files are present and read during execution, if-statements with substring checks might appear and be relevant. Future research could broaden the evaluation to investigate this possibility.

To further address these limitations, future work could extend the seeder to support a broader range of input types, such as `int`, `float`, and `bool`. Additionally, indirect uses of input parameters—such as assigning an input's length to a variable and later using that variable in a condition—could be detected through static data flow analysis. This would allow the seeder to trace parameter influence through intermediate transformations or assignments, including manipulations where the parameter is reassigned or modified before use. Such capabilities would enhance the precision of the seeding process and likely increase its applicability to more complex, real-world codebases.

9 Conclusion

This work presents a lightweight extension to Pynguin that enhances search-based test generation through string constraint-guided seeding. The extension targets a known limitation of genetic algorithms—difficulty in handling string-based conditions—by seeding the initial population with literals extracted from `if`-statements in the code's abstract syntax tree. It integrates seamlessly with Pynguin's workflow by operating fully in-memory and preserving low initialization cost.

While the evaluation showed partial improvements, especially in controlled internal tests, broader applicability was constrained by the structural characteristics of many functions, such as lacking string-typed parameters or using indirect or unsupported logic. Addressing these limitations will require expanding the seeding mechanism to handle more complex input types and diverse constraint patterns.

Collectively, these findings show that by adapting seeding strategies established in Java-based SBST to the dynamic nature of Python, this extension demonstrates how targeted, efficient enhancements can strengthen existing test generation tools. Looking ahead, such improvements offer a path toward reducing the manual burden of test creation and bringing the full benefits of automated testing—low effort, early integration, and improved software quality—closer to widespread realization.

References

1. Abbas Albadr, M., Tiun, S., Ayob, M., Al-Dhief, F.: Genetic algorithm based on natural selection theory for optimization problems. Symmetry **12**(11), 1758 (2020). https://doi.org/10.3390/sym12111758
2. Committee, T.C.: Sbst 2024 tool competition: Klara. https://github.com/usagitoneko97/klara (2024), accessed July 2025
3. Erni, N., Mohammed, A.A.M.A., Birchler, C., Derakhshanfar, P., Lukasczyk, S., Panichella, S.: Sbft tool competition 2024 – python test case generation track. arXiv preprint arXiv:2401.15189 (2024), https://arxiv.org/abs/2401.15189v1, to appear in Proceedings of the 17th International Workshop on Search-Based and Fuzz Testing (SBFT@ICSE 2024)
4. Graebner, P.J.G.: Pynguin documentation. https://pynguin.readthedocs.io/en/latest/ (2024), accessed July 2025
5. Graebner, P.J.G., Fraser, G.: Automated unit test generation for python. Empirical Software Engineering **27**(6), 1–35 (2022). https://doi.org/10.1007/s10664-022-10248-w
6. Khelladi, D.E., Reux, C., Acher, M.: Unify and triumph: Polyglot, diverse, and self-consistent generation of unit tests with llms (2025), https://arxiv.org/abs/2503.16144
7. Lukasczyk, S., Kroiß, F., Fraser, G.: An empirical study of automated unit test generation for python. Empirical Software Engineering **28**(1), 36 (2023). https://doi.org/10.1007/s10664-022-10248-w
8. Lukasczyk, Stephan: Pynguin: Python general unit test generator. https://github.com/se2p/pynguin/tree/0.41.0 (2025). https://doi.org/10.5281/zenodo.15496289, gitHub Repository
9. Rojas, J.M., Fraser, G., Arcuri, A.: Seeding strategies in search-based unit test generation. Software Testing, Verification and Reliability **26**(5), 366–401 (2016). https://doi.org/10.1002/stvr.1585
10. Samuel Brinkmann, Trung Kien Hoang, Levin Schulz, and Sharui Yang: Penguin and seeding. https://github.com/Priapos1004/penguin_and_seeding (2025), gitHub Repository
11. Samuel Brinkmann, Trung Kien Hoang, Levin Schulz, and Sharui Yang: Penguin and seeding. https://github.com/Priapos1004/penguin_and_seeding/compare/v0.0.1...v0.1.0 (2025), gitHub Repository Compare v0.0.1...v0.1.0
12. Sheskin, D.J.: Handbook of Parametric and Nonparametric Statistical Procedures. Chapman & Hall/CRC, Boca Raton, FL, 5th edn. (2011). https://doi.org/10.1201/9780429186196
13. Team, S.T.C.: Utbotpython: Symbolic execution meets fuzzing for python. https://github.com/UnitTestBot/UTBotPython (2024), accessed July 2025
14. TheAlgorithms: TheAlgorithms/Python: All Algorithms implemented in Python. https://github.com/TheAlgorithms/Python (2025), gitHub repository, accessed 17 July 2025
15. Vaddadi, S.A., Thatikonda, R., Padthe, A., Arnepalli, P.R.R.: Shift-left testing paradigm process implementation for quality of software based on fuzzy. Soft Computing **27**(14), 10001–10019 (2023)
16. Wang, Y., Mäntylä, M., Liu, Z., Markkula, J.: Test automation maturity improves product quality: Quantitative study of open source projects using continuous integration. arXiv preprint arXiv:2202.04068 (2022), peer-reviewed version published in Empirical Software Engineering (doi:10.1007/s10664-022-10259-7)

Appendix

Use of Language Tools

ChatGPT4o and Grammarly were used for grammar and spelling checks in this paper.

Contribution of Group Members

While all group members collaborated closely and participated in regular code reviews throughout the project, each person had specific areas of focus:

Samuel Brinkmann led the integration of the new module into the existing Pynguin framework. He also contributed significantly to the overall project, including the evaluation of the module, the refinement of the seeding strategy, and the writing of this report.

Trung Kien Hoang was primarily responsible for designing and implementing the internal tests that guided the development of the seeding strategy. He also contributed to the broader evaluation process.

Levin Schulz and Sharui Yang focused on the conception and implementation of the seeding strategy itself, including the AST traversal and extraction mechanisms.

Despite these individual emphases, the project was carried out as a joint effort, with shared decision-making and mutual support across all components.