Evaluating Feedback-Directed Random Test Generation

Alex Bruma
Simon Fraser University

Ekam Sidhu
Simon Fraser University

Zhengying Sun
Simon Fraser University

Abstract

Unit tests are vital but slow to write and edge cases still slip through. We revisit feedback-directed random testing with a compact, modular generator. The tool grows sequences of method calls, runs them, and keeps only those that behave, using light contracts and filters to prune crashes. One simple parameter (the probability of reusing earlier sequences) lets you guide exploration toward deeper object states without heavy analysis. The output is plain JUnit regression tests with basic assertions and no special runtime, so suites stay readable and easy to run.

The goal is practicality: minimal setup (a class to test and a time budget), predictable behavior, and a design that's easy to extend. The result is a clear baseline for feedback-directed testing that balances simplicity, speed, and coverage while leaving room for richer contracts or advanced techniques.

1 Introduction

Writing comprehensive unit tests is a critical yet time-consuming task in software development. Developers often need to manually craft test inputs to cover diverse control paths, which can be both error-prone and tedious. Manual test case design suffers from what might be characterized as scalability limitations and typically fail to expose edge-case bugs or unexpected behaviors in complex systems [1]. Within this broader analytical framework, automated test generation has garnered substantial attention due to its ability to enhance coverage, and reduce human effort. Techniques in this area include symbolic execution, search-based approaches, mutation testing, and random testing [1].

Among these techniques, feedback-directed random test generation has emerged as an effective middle ground between pure randomness and expensive analysis. Pacheco *et al.* introduced Randoop, a system that incrementally builds sequences of method calls and uses runtime feedback to decide which sequences are worth extending [2]. This approach avoids generating tests that throw exceptions, misuse null values, or repeat identical object constructions, enabling more efficient and targeted test exploration.

Our project aims to reimplement the core idea behind Randoop, focusing on building a lightweight and modular feedback-directed test generator. By enabling the automatic construction of test suites that achieve high statement coverage with minimal user effort, we hope to provide evidence that supports both the practicality and effectiveness of this approach.

To evaluate our system, we compare it against Randoop itself. Through this comparison, we investigate whether a simplified, re-implemented version of Randoop can provide comparable results.

2 Technique

This section describes our technique for implementing feedback-directed random test generation.

2.1 Statement Sequences

Each unit test is created from a sequence of statements, where each statement is either a method invocation, constructor invocation, or a primitive value. The results of these statements are saved for use as parameters in subsequent statements; the result of method call and constructor call statements are

their return values, while the result of primitive value statements are the primitive values themselves.

While primitive value statements are simple, and consist solely of their result, method and constructor call statements consist of the method/constructor's name alongside a list of its parameters. If a method is not static, the first entry in its parameter list will be its receiver, while the remaining parameters will be its normal input arguments.

2.2 Statement Generation

The main component of sequences are method invocation statements, the other statement types are only created when their result is needed in the parameter list of another statement.

Method invocation statements are created by first choosing a random public method from the input class(es) to invoke, where less-used methods have a higher probability of being selected. Then, a statement calling that method is constructed by acquiring its required parameters, including a receiver for non-static methods, from the results of previously created statements.

The generator will first look for usable statements in the current sequence, then, if none are found, it will check other previously created sequences. If a usable statement is found only in a separate sequence, that sequence will be concatenated with the current one, such that the result of the statement it contains can be utilized in the currently forming method invocation. Alternatively, if no applicable statements are found, a newly generated statement will be appended to the sequence, whose type is determined as follows:

- A constructor invocation statement will be created if an instance of a class is needed, whether it be to serve as a method receiver or an argument. In doing so, a public constructor of the required class is randomly selected, and its arguments (if any) are resolved in the same way as described above. Then, the constructor is executed to acquire the class instance; if this execution causes an exception, the creation process restarts from the beginning.
- A primitive value statement will be created if a primitive, or a simple object (e.g. an array), is needed as an argument.

Algorithm 1 Feedback-directed sequence generation algorithm

```
1: function GenerateSequences(classes, timeLimit, maxSequences, reuseProb)
2:     validSeqs, invalidSeqs, seenSeqs, seqPool ← ∅
3:     while within timeLimit and |seenSeqs| < maxSequences do
4:         seq ← randOrNewSeq(validSeqs, reuseProb)
5:         cls ← getRandomClass(classes)
6:         m ← getRandomPublicMethod(cls)
7:         args ← getArgsForMethod(seqPool, m)
8:         append newStmt(m, args) to seq
9:         if violatedFilters(seq) then
10:            continue
11:        end if
12:        execute(seq)
13:        if threwException(seq) or violatedContracts(seq) then
14:            insert seq into validSeqs
15:            insert seq into seqPool
16:        else
17:            insert seq into invalidSeqs;
18:        end if
19:        insert getFingerprint(seq) into seenSeqs
20:    end while
21:    return (validSeqs, invalidSeqs)
22: end function
```

Additionally, there is a small chance that any given parameter will be either a completely random value or null, regardless of any existing statements. This helps increase the variability of generated sequences, so as to better explore edge-case behaviour.

Once all of the needed parameters have been acquired, the generator will append the method call statement to the end of the current sequence, and execute it to get its result, thereby completing the creation process.

2.3 Generation Algorithm

Algorithm 1 illustrates how sequences are generated, and is largely inspired by the algorithm used in [2].

Although the generation algorithm can take numerous arguments, the only required input is the list of classes, as all other arguments have default values (being 2 minutes, 1000, and 0.75 respectively)

As shown on line 4, in each iteration, the algorithm first decides whether to extend an existing sequence, or start from an empty one. While the first iteration will always use an empty sequence, as there aren't any existing sequences to extend, all other iterations will have a probability given by *reuseProb* to extend an existing sequence.

Lines 5 to 8, and line 12 describe how a new method invocation is created and appended to the selected sequence, as described in Subsection 2.2. There is an addition to this however, on line 9, before the sequence is executed, the sequence is checked against filters. This simply removes duplicate sequences by checking whether the current sequence is in the set of previously seen sequences (*seenSeqs*).

Then, on line 13 the sequence is checked both against contracts, and for thrown exceptions during execution. Contracts are assertions that ensure all the objects and method invocations within the sequence are valid. These ensure that `o.equals(o)` returns true for non-null objects, and that none of `o.equals(o)`, `o.hashCode()`, `o.toString()` throw exceptions for non-null objects.

If the sequence passes both the exception and contract checks, it is considered valid and stored in both the set of valid sequences and the sequence pool (lines 11–12). Otherwise, it is saved in the set of invalid sequences (line 14).

Finally, a fingerprint (string representation) of the sequence is stored (line 16) in the set of seen sequences to filter out duplicates.

2.4 Code Generation

Once the sequence generation is complete, each sequence is converted into its own JUnit regression test.

All sequences convert their statements in order, and every converted statement, other than void method invocations, has their result assigned to a variable for inclusion in subsequent statements which utilize that result. Contract assertions, such as `o.equals(o) == true`, are also included at the end of each test for all non-null, non-primitive objects, to ensure all contracts remain satisfied as the code is modified.

3 Evaluation and Analysis

3.1 Objectives and Metrics

Our primary goal is to measure the effectiveness of our RANDOOP re-implementation. We track:

1) **Instruction coverage** (statement coverage) via JaCoCo.

2) **Predicate coverage** (branch coverage) via JaCoCo.

These metrics are stricter than the basic-block coverage used in the original study by Pacheco et al. [2]. For a fair comparison, we obtained the latest version of Randoop [3] (released June 6, 2025) and executed it under the same testing environment.

3.2 Experimental Setup

- **Subjects**: `TreeMap`, `FibHeap`, `BinTree`, `BinomialHeap`. From the original Randoop paper [4].
- **Environment**: OpenJDK 21.0.3, Ubuntu 24.04, Maven 3.9.6, JaCoCo 0.8.11.
- **Time**: less than 30 s
- **Sequence reuse probability** p_{reuse}: 0.75 - 0.99, depending on the test class.

3.3 Results

Let's examine the results when comparing our feedback-directed test generation implementation to Randoop. Overall, our test suite performed quite well, and notably, our generator is, on average, significantly faster at producing a comparable number of tests while achieving similar coverage. This performance advantage is likely due to the simplicity of our implementation, as Randoop's codebase is substantially larger and more complex than ours.

Table 1: Coverage for `TreeMap`.

Metric	Original Randoop	This work
Instruction coverage	68%	60%
Predicate coverage	61%	57%
Generation time (s)	25	8.2
Test-suite size	500	1600
p_{reuse}	-	0.99

Table 2: Coverage for `BinTree`.

Metric	Original Randoop	This work
Instruction coverage	100%	96%
Predicate coverage	100%	96%
Generation time (s)	7.1	6.4
Test-suite size	413	500
p_{reuse}	-	0.75

Table 3: Coverage for `FibHeap`.

Metric	Original Randoop	This work
Instruction coverage	55%	75%
Predicate coverage	68%	80%
Generation time (s)	25.2	6
Test-suite size	500	200
p_{reuse}	-	0.85

Table 4: Coverage for `BinomialHeap`.

Metric	Original Randoop	This work
Instruction coverage	82%	87%
Predicate coverage	82%	88%
Generation time (s)	9	6.1
Test-suite size	944	400
p_{reuse}	-	0.85

3.4 Discussion

For every subject class we adopted an empirical, "parameter search" approach: we repeatedly executed our generator while going through its most impactful parameters and retain the setting that maximised coverage. Two parameters emerged as dominant:

1. Suite cardinality: The total number of test methods created in the generated JUnit class.
2. Sequence-reuse probability p_{reuse}: The likelihood that construction of a new sequence begins with an existing, already-validated sequence rather than from scratch.

Effect of p_{reuse}. Raising p_{reuse} biases the algorithm toward *depth* rather than *breadth*: the average sequence grows longer because nearly every new test splices additional calls onto a previously successful prefix. The practical consequence is a test suite whose individual methods explore progressively richer object states.

This depth-first tendency is especially beneficial for APIs that expose a large public surface but guard critical behaviour behind private helper methods. TreeMap exemplifies that pattern: it offers dozens of public operations whose internal implementations cascade through a maze of private balancing and ordering routines. Short, one-shot sequences rarely use the TreeMap enough to trigger those private methods. In contrast, the longer traces produced with $p_{\text{reuse}} \approx 1$ steadily mutate the structure, performing inserts, deletes, and key rotations in succession until the data structure reaches the edge cases that call into its otherwise "rare" private functions. Each such indirect invocation lights up untouched instructions and additional branch alternatives, ultimately lifting both statement and predicate coverage.

Empirical outcome. Across much parameter tweaking we observed a relationship for `TreeMap`: as soon as p_{reuse} exceeded 0.9, coverage climbed sharply; our best run, with $p_{\text{reuse}} = 0.99$ and a capped suite size of 1600 methods, achieved the 60 instruction and 55 branch coverage reported in Table 1. In contrast, the other three data-structure subjects: `BinTree`, `FibHeap`, and `BinomialHeap` derive most of their complexity from recursive algorithms reachable via

short call chains, so they displayed diminishing returns beyond $p_{\text{reuse}} \approx 0.80$. These findings reinforce a general heuristic: use a high sequence-reuse probability for classes whose interesting states require many interdependent operations, and a lower value when the behaviour of interest sits close to the constructor.

4 Future Directions

Our present work aims to reimplement the core feedback-directed random test generation algorithm, which is a foundational component of Randoop. While the implementation achieves the basic goal of generating test suites that increase statement coverage, there do exist potential extensions to improve its fault detection ability.

The analysis by Arasteh *et al.* [5] supports the use of heuristic algorithms—such as the Shuffled Frog-Leaping Algorithm (SFLA), Imperialist Competitive Algorithm (ICA), and Artificial Bee Colony (ABC)—which can improve test-data generation efficiency by leveraging branch and predicate distance metrics. Integrating such heuristics into our feedback loop could systematically steer input generation toward satisfying complex path conditions, particularly for branches that pure random exploration rarely reaches.

The investigation by Xing *et al.* [6] introduces a hybrid intelligent search approach (BB-HC) that combines branch-and-bound with hill climbing, enhanced by interval arithmetic, to solve path constraints more efficiently. By integrating a lightweight version of BB-HC, we could pre-compute promising input ranges for difficult branches and use them as seeds during random sequence generation. This hybrid approach would retain Randoop's flexibility while systematically unlocking deep execution paths.

Pursuing these directions would evolve our current prototype into a more adaptive and fault-oriented automated testing framework, balancing structural adequacy, behavioral diversity, and maintainability.

5 Related Work

There are also other programs that can be useful for automatic test generation. Some such programs are described below.

GRT (Guided Random Testing).
GRT [7] is an extension of basic feedback-directed test generation that utilizes both static and dynamic analysis to maximize code coverage. Initially, it does static analysis of the code being tested to gather information about the program, such as what constants exist, as well as method dependencies and side-effects. It uses this knowledge to create a pool of primitive values that can be used when generating test inputs, and determine the properties of individual methods to understand how they can be sequenced together in tests. Then, at runtime, GRT combines this information with dynamic feedback, including data on the types of objects at runtime and current test coverage, to help construct objects to use as input for further tests, and determine which methods to generate more tests for. Such methods could also be used to improve this work's effectiveness, as stated in Section 4.

Search-based Test Generation.
Tools for search-based test generation utilize fitness functions to maximize code coverage. A notable tool that uses this method of test generation is EvoSuite. EvoSuite, by default, uses a search algorithm called DynaMOSA (Dynamic Many-Objective Sorting Algorithm) [8] to generate test cases. It allows users to choose multiple coverage criteria to optimize simultaneously, and is very effective in doing so. [9]

6 Conclusion

Our re-implementation of feedback-directed random test generation proved to be effective and competitive with the original Randoop. While our tool is much smaller and simpler in design, it was able to generate test suites with similar coverage in a fraction of the time. The evaluation showed that careful tuning of parameters, especially sequence-reuse probability, has a clear impact on coverage, with high reuse

values benefiting classes that require longer interaction chains to reach complex states. These results suggest that even a lightweight generator can deliver strong performance if it focuses on efficient sequence construction and uses feedback to guide exploration. Overall, the work demonstrates that a simpler approach can still achieve meaningful results, and provides a solid base for future improvements.

References

[1] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42, 2012. doi:10.1109/IWAST.2012.6228988.

[2] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84, 2007. doi:10.1109/ICSE.2007.37.

[3] Randoop Project. Randoop manual, 2025. `https://randoop.github.io/randoop/manual/index.html`, Accessed August 2025.

[4] Randoop Project. Randoop issta 2006 test inputs, 2025. `https://github.com/randoop/randoop/tree/master/src/testInput/java/randoop/test/issta2006`, Accessed August 2025.

[5] Parisa Arasteh, Mohammad Fazel Zarandi, and Mohammad Reza Meybodi. Constraint-based heuristic algorithms for software test generation. *Applied Soft Computing*, 93:106320, 2020. doi:10.1016/j.asoc.2020.106320.

[6] Ying Xing, Yun-Zhan Gong, Ya-Wen Wang, and Xu-Zhou Zhang. A hybrid intelligent search algorithm for automatic test data generation. *Mathematical Problems in Engineering*, 2015:Article ID 617685, 15 pages, 2015. doi:10.1155/2015/617685.

[7] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. Grt: Program-analysis-guided random testing (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 212–223, 2015. doi:10.1109/ASE.2015.49.

[8] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018. doi:10.1109/TSE.2017.2663435.

[9] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, Jose Campos, and Annibale Panichella. Evosuite at the sbst 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 28–29, 2021. doi:10.1109/SBST52555.2021.00012.