# Software Improvement with Gin: a Case Study

Justyna Petke[1] and Alexander E.I. Brownlee[2]

[1] University College London, UK
j.petke@ucl.ac.uk
[2] University of Stirling, UK
sbr@cs.stir.ac.uk

**Abstract.** We provide a case study for the usage of Gin, a genetic improvement toolbox for Java. In particular, we implemented a simple GP search and targetted two software optimisation properties: runtime and repair. We ran our search algorithm on Gson, a Java library for converting Java objects to JSON and vice-versa. We report on runtime improvements and fixes found. We provide all the new code and data on the dedicated website: `https://github.com/justynapt/ssbseChallenge2019`.

**Keywords:** Genetic Improvement · Search-Based Software Engineering

## 1 Introduction

Genetic improvement (GI) uses automated search to improve existing software [10]. GI-evolved changes have already been incorporated into development [6,7]. Only recently two frameworks emerged that aim to help researchers experiment with GI: Gin [3,11] and PyGGI [1,2]. The new version of Gin provides support for large-scale Java projects, thus we decided to use it for our study.

In this paper we aim to determine whether we can improve various software properties using Gin. We chose Gson for the case study, as it is written in Java and follows the Maven directory structure. The second version of Gin provides utilities for setting up Maven and Gradle projects, so that the user only needs to provide project name and top level directory. This way researchers can quickly test their novel GI strategies on such projects. Several of Gin's utilities are method-based, thus search can be restricted to individual methods.

In this work we use Gin to generate patches for one of the most frequently used methods of Gson. We aim to improve its runtime and fix (injected) bugs.

## 2 Subject Program

Gson[3] is a Java library for converting Java Objects to JSON and vice-versa. It is used by over 152,000 projects on GitHub, and there have been 39 releases so far. It can be built with Maven or Gradle, and follows the standard project structure. In this work we use the latest release, that is, gson-parent-2.8.5.

[3] https://github.com/google/gson

We first ran cloc[4] and the PIT mutation tool[5] to get information about the project. Gson contains 50874 lines of code, 25193 of which are in Java. The test suite achieves 83% line coverage and 77% mutation coverage.

# 3 Test Suite

The test suite consists of 1051 JUnit tests, 1050 of which are runnable with `mvn test` (1 test is skipped; the total runtime of the remaining 1050 is < 10 seconds). Running PIT issued warnings that two tests (`com.google.gson.functional.ConcurrencyTest.testMultiThreadDeserialization` and `com.google.gson.functional.ConcurrencyTest.testMultiThreadSerialization`) leave hanging threads. We ran those tests using Gin's utility, `gin.util.EmptyPatchTester` (which runs all provided unit tests in the input file for a project), and indeed the program did not terminate, unless the `-j` option was added, which runs tests in a separate JVM. Therefore, we fixed those tests by adding a shutDown hook for the ExecutorService instances at the end of each of the two faulty unit tests[6].

# 4 Methodology

We set out to show how the latest version of the genetic improvement tool, Gin [3], can be used for the purpose of runtime improvement as well as program repair. Therefore, we used the same search algorithm for targeting both objectives: genetic programming; the most frequently used strategy in genetic improvement [10]. Each individual in the population is represented as a list of source code edits.

## 4.1 Search

Following the famous GenProg algorithm structure [8], for each generation we select two parents from the previous population at random, apply 1-point crossover to create two children, and append both parents and both children to the current population. If the required population size is not divisible by four, we add the original program to the population until we reach the desired number[7]. Finally, we mutate each of the created individuals and calculate their fitness.

*Crossover:* Crossover takes two parents, i.e., a list of edits, and creates two children: one comprising the first half of edits of parent 1 and the second half of edits of parent 2; the second child containing the remaining edits.

[4] https://cloc.org/

[5] Plugin used: https://github.com/STAMP-project/pitmp-maven-plugin

[6] Fixed test class is available on the submission's website, in the input folder.

[7] We decided to make this small change following insight that fixes usually require no more that four AST node edits [9].

*Mutation:* We use two types of mutation operators, which were introduced in Gin [3]. The first type are constrained statement edits, that contain DELETE, COPY, SWAP and REPLACE operations to adhere with the Java grammar. DELETE simply targets a single Java statement for deletion. The remaining three edits target matching pairs of Java statements (e.g. two assignment statements, or two `if` statements). The second mutation type are Binary and Unary replacement operators. These follow the micro-mutations in [5]: binary operator replacement will replace e.g. `==` with `!=`, or `<` with `>`; unary operator replacement will replace e.g. `++` with `--`.

*Fitness:* For the purpose of runtime improvement, we simply used runtime measured by the system clock, in milliseconds, as fitness. We only allowed individuals that pass all the tests to be considered for mating in the next generation. For the purpose of program repair, we used the number of tests failed as a fitness measure. We only allowed individuals that compile and do not fail more tests than the original program to move to the mating population.

### 4.2 Setup

For our experiments, due to time constraints, we used 10 generations with population size of 21. For runtime improvement, we ran each test (with 2 sec. timeout) 500 times and took the total time. This is to off-set the fact that each test case can be run in milliseconds. For program repair this condition is not necessary.

In order to establish which methods to improve, we first ran Gin's utility `gin.util.Profiler` to establish which methods are the most frequently used. This utility uses hprof[8] to check how often a method appears on the call stack, sampling it every 10ms. This is a non-deterministic procedure, so we ran each test 10 times to ensure the most frequently used methods are in the output file.

We implemented 4 new classes: `gin.util.GP` is an abstract class, which also processes input and output; `gin.util.GPSimple` implements GP search; while `gin.util.GPFix` and `gin.util.GPRuntime` extend it, implementing fitness functions.

We ran our experiments on a Lenovo ThinkPad Edge laptop with Intel Core i5-2410M CPU @ 2.30GHz 4 processor, running 64-bit Ubuntu 18.04.2 LTS.

## 5 Results

Gin's `Profiler` revealed that the most frequently used method is `com.google.gson.Gson.newJsonReader`. However, we did not use it in our experiments as it consisted of only three lines that essentially just instantiated `JsonReader` and returned it. Thus this method is unlikely to be improvable. Therefore, we opted to target the second most frequently used method: `com.google.gson.GsonBuilder.create`. This method also contains two addition operators, so it would be interesting to see if the Binary operator could find improvements. `Profiler` identified 78 tests that cover this method. Overall, `Profiler` found 585 tests on the hprof call stack (sampled at 10ms intervals, so not all 1050 tests captured, as expected).

[8] `https://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html`

### 5.1 Runtime improvement

We first tried the constrained statement edits. The GP run finished in 22 minutes. 56 improved patches were found. The best individual found improved runtime by 19% on the training set. The best patch found removed one line: `addTypeAdaptersForDate(datePattern, dateStyle, timeStyle, factories);`[9] However, when the mutant was run with `mvn test`, several test cases failed. This shows that the methods used were not enough to capture the desired software behaviour for that method.

We can attribute this to the process the `Profiler` uses to determine the tests associated with a given target method. The `Profiler` samples the call stack at regular intervals, so could conceivably miss some calls. It also excludes parametrised tests. Running on a subset of the tests helps avoid overfitting to any dominant tests in the complete set, but cannot guarantee correct behaviour (insofar as the test suite can measure it). The only solution to this is to treat the limited test set produced by the `Profiler` for a target method as a quick-running surrogate for the whole test set, but one should still evaluate on the whole set at intervals.

We thus ran a second experiment, with all 585 tests identified by the `Profiler` for training. Since we increased the number of tests to 585 from 78, we decreased the number of repetitions of each test by the same fraction (500 / 7.5 = 67), to save time. The experiment finished in 40 minutes, finding 46 improved patches.

This time Gin was able to find improvements that generalised to the whole test suite of 1050 tests. The best patch improved runtime by 24% on the training set and swapped the following two lines:

```
factories.addAll(hierarchyFactories);
addTypeAdaptersForDate(datePattern, dateStyle, timeStyle, factories);
```

That being said, the improvement is unnoticeable, when tests are run once. This is due to the fact that all tests run in a fraction of a second (1050 tests finish in less than 10 seconds). Moreover, the 24% improvement amounts to 1.25 seconds, which could be due to environmental bias, as total execution time was calculated for fitness evaluation purposes.

Even though significant improvements have not been found, there is a key point worth noting. The test suite has a strong impact on the validity of the results of a GI framework. Despite the evolved patch passing all tests, we doubt that this was the intended behaviour of the software, as the `addTypeAdaptersForDate` method uses the `factories` variable. By swapping the statements, the value of the `factories` variable is changed. Thus it is crucial to re-run the full test suite at regular intervals during the search, and before application of GI, it is important to ensure that the test suite is adequate. Generating additional tests on the correctly running original version of the program using a tool such as EvoSuite [4] is also advised.

[9] This mutant can be obtained by running `gin.PatchAnalyser` with the text for the patch found in the output file of GPRuntime.

### 5.2 Repair

In order to inject faults, we looked at the PIT reports. We found one mutation that was both killed by the existing test suite and could be potentially found by our mutation operators, that is, a change of sign from +3 to −3. We thus introduced that mutant and ran our experiment with the 78 tests identified by `Profiler` to cover the `GsonBuilder.create` method. Since we did not repeat test runs, this experiment was quick, running in under 2 minutes.

The original code segment affected by the mutation was as follows:

```
List<TypeAdapterFactory> factories = new ArrayList<TypeAdapterFactory>(
this.factories.size()+ this.hierarchyFactories.size()+ 3);
```

The mutant looked like this:

```
List<TypeAdapterFactory> factories = new ArrayList<TypeAdapterFactory>(
this.factories.size()+ this.hierarchyFactories.size()- 3);
```

In this case, only the Binary and Unary replacement mutations were used in the GP. Given that we ran the process with 21 individuals and 10 generations, 210 patches were generated overall during the search. 171 of those passed all the tests. The first patch was found in the first generation, and changed the minus sign to a multiplication. This patch passed all 1050 tests. In this case the fix was found quickly because of the limited search space: having limited the possible code mutations that the GP could explore to only mutations that would be likely to fix the bug. To fix a wider range of bugs (i.e. where we do not know the bug *a priori*), the number of edit types would need to be extended to at least the wider range included with Gin and the search would take correspondingly longer.

As in the runtime experiment, we observed that the 78 tests might not be enough to cover all the behaviour, we ran another experiment with the 585 tests identified by the `Profiler`. This time 174 fixes were found in 4 minutes. This set contained several individuals that contained the required mutation that changed the minus to the plus sign. However, again the first patch found changed minus to a multiplication instead. The question arises whether the fixes found are true fixes, or whether the test suite should be improved.

We also injected another fault, that swaps the following two statements:

```
factories.addAll(this.factories);
Collections.reverse(factories);
```

Out of 78 tests, just one failed for this mutant. GP search took 41 seconds, this time limited to the constrained statement mutations. No fix was found. We also ran this experiment with 585 tests, to avoid overfitting. No fix was found either. We know the fix is in the search space, so a larger run could potentially produce the desired fix (or different random seeds for mutation selection and individual selection). Given that previous research found that fixes usually contain short mutations, perhaps a different search strategy would have been more effective. The current one almost always increases the size of each mutant by one.

Finally, we introduced a bug that copied the following line right under itself:

```
Collections.reverse(factories);
```

We ran GP with the 78 tests and 585 tests, as before. In both cases the correct fix was found in the first generation (i.e., deleting the extra line).

## 6 Conclusions

We showed how Gin can be used for the purpose of program's runtime improvement and repair. It shows how quickly and easily researchers can conduct GI experiments on large Java projects. We added a simple GP search to Gin, and applied it to Gson. Our results show that expression-level changes are possible with Gin that can lead to useful mutations (fixes). There are several future directions. More fine-grained fitness values are possible with Gin, as it captures the expected and actual result of tests. This could guide the search better. From our results a question arises whether GP is the best approach for GI.

We also showed that existing test suites are not enough to capture software behaviour. We pose that Gin can thus be used to test the strength of a given test suite. Gin also provides a utility to generate EvoSuite tests, which could strengthen the test suite, though currently the feature is experimental.

All data for replicability purposes is available on the dedicated website: `https://github.com/justynapt/ssbseChallenge2019`.

*Acknowledgements* The work was funded by the UK EPSRC grant EP/P023991/1 and Carnegie Trust grant RIG008300.

## References

1. An, G., Blot, A., Petke, J., Yoo, S.: PyGGI 2.0: Language independent genetic improvement framework. In: ESEC/FSE. ACM (2019)
2. An, G., Kim, J., Yoo, S.: Comparing line and AST granularity level for program repair using PyGGI. In: Proc. Intern. GI Workshop @ICSE. pp. 19–26. ACM (2018)
3. Brownlee, A.E.I., Petke, J., Alexander, B., Barr, E.T., Wagner, M., White, D.R.: Gin: Genetic improvement research made easy. In: GECCO. ACM (2019)
4. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Proc. ACM SIGSOFT symposium and the European conf. on Foundations of software engineering. pp. 416–419. ACM (2011)
5. Haraldsson, S., Woodward, J., Brownlee, A., Cairns, D.: Exploring fitness and edit distance of mutated python programs. In: EuroGP. pp. 19–34. Springer (2017)
6. Haraldsson, S.O., Woodward, J.R., Brownlee, A.E.I., Siggeirsdottir, K.: Fixing bugs in your sleep: how genetic improvement became an overnight success. In: GECCO, Companion Material Proc. pp. 1513–1520. ACM (2017)
7. Langdon, W.B., Lam, B.Y.H., Petke, J., Harman, M.: Improving CUDA DNA analysis software with genetic programming. In: Proc. of the GECCO, GECCO. pp. 1063–1070. ACM (2015)
8. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Trans. Software Eng. **38**(1), 54–72 (2012)
9. Martinez, M., Monperrus, M.: Mining software repair models for reasoning on the search space of automated program fixing. EMSE **20**(1), 176–205 (2015)
10. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: A comprehensive survey. IEEE Trans. Evolutionary Computation **22**(3), 415–432 (2018)
11. White, D.R.: GI in no time. In: GECCO, Companion Material Proc. pp. 1549–1550. ACM (2017)