Gin: Genetic Improvement Research Made Easy

Alexander E.I. Brownlee
Computing Science and Mathematics
University of Stirling
Stirling, UK
sbr@cs.stir.ac.uk

Justyna Petke
Department of Computer Science
University College London
London, UK
j.petke@ucl.ac.uk

Brad Alexander
School of Computer Science
University of Adelaide
Adelaide, Australia
brad@cs.adelaide.edu.au

Earl T. Barr
Department of Computer Science
University College London
London, UK
e.barr@ucl.ac.uk

Markus Wagner
School of Computer Science
University of Adelaide
Adelaide, Australia
markus.wagner@adelaide.edu.au

David R. White
Department of Computer Science
The University of Sheffield
Sheffield, UK
d.r.white@sheffield.ac.uk

ABSTRACT

Genetic improvement (GI) is a young field of research on the cusp of transforming software development. GI uses search to improve existing software. Researchers have already shown that GI can improve human-written code, ranging from program repair to optimising run-time, from reducing energy-consumption to the transplantation of new functionality. Much remains to be done. The cost of re-implementing GI to investigate new approaches is hindering progress. Therefore, we present Gin, an extensible *and* modifiable toolbox for GI experimentation, with a novel combination of features. Instantiated in Java and targeting the Java ecosystem, Gin automatically transforms, builds, and tests Java projects. Out of the box, Gin supports automated test-generation and source code profiling. We show, through examples and a case study, how Gin facilitates experimentation and will speed innovation in GI.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**; **Search-based software engineering**;

KEYWORDS

Genetic Improvement, GI, Search-based Software Engineering, SBSE

1 INTRODUCTION

Genetic improvement (GI) is a young field of software engineering research that uses search to improve existing software. GI aims to improve both functional, notably bug fixing, and non-functional properties of software, such as runtime or energy consumption. The intersection of automated program repair (APR) and GI has had the greatest impact to date, from the release of the GI-based tool GenProg [27] to successful integration of APR into commercial development processes [19, 20]. Non-functional improvement (NFI) is the branch of GI that, as its name suggests, improves non-functional properties without, in contrast to APR, needing an implicit specification or a user-provided test oracle, since it can use its input program as its functional oracle. NFI has also had significant industrial impact — BaraCUDA, a widely used sequence mapping tool, accepted GI-evolved patches in 2015 [24].

GI abounds with open problems. GI searches the space of program variants created by applying mutation operators. The richness of this space depends on the power and expressivity of the mutation operators; we have not yet identified mutation operators that simultaneously define a rich and dense search space. Given a set of operators, the GI space is usually vast and sparsely populated by variants that meet a specification or that a human might write. Efficiently traversing GI spaces under a resource bound remains an open problem. A key subproblem here is how to efficiently integrate testing program variants into the search.

Working to close these problems requires experiment-driven innovation; experimentation necessitates engineering, some novel, but much that is not. The time researchers currently take to build a GI substrate — either writing from scratch or finding, adapting, and binding together existing tools — involves reimplementing many wheels, like parsing and program transformation. This is because existing work relies on bespoke tools that are not designed for reuse or modification. For example, some tools require expertise in programming languages, such as Lisp [37], that many software engineering researchers do not often use. The lack of shared tooling is hampering GI research, especially into NFI; it hinders reproducability and slows innovation.

The potential benefits of a shared, tooling substrate for GI experimentation are enormous. We need look no further than the impact such tooling has had on other areas of computer science. The Evolutionary Computation Library ECJ [29] is a general-purpose, *extensible* framework for evolutionary computation (EC); anecdotally, its release facilitated experimentation and reproduction in EC [29]. SimpleScalar [9] is an open source set of tools for simulation of modern processor architectures. Prototyping processors in hardware is simply prohibitive for most academics; SimpleScalar is the simple and efficient testbed on which academic research in computer architecture rested for over a decade [17]. A more recent example is Google's TensorFlow[5], a library for numerical computation and large-scale machine learning. It has democratized machine learning, leading to an explosion of papers[1] and, anecdotally [7], providing a key capability for some AI startups.

To reproduce this success and accelerate research in GI, we introduce Gin, an experimental substrate for GI. We have instantiated Gin in Java for the Java ecosystem. We chose Java to facilitate the application of GI to a prominent object-oriented language and because Java is a lingua franca for software engineers, so its adoption

[1] Over 8000 citations in Google Scholar for the cited article.

The definitive version was published in *GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO 2019: The Genetic and Evolutionary Computation Conference, Prague, Czech Republic, 13.07.2019-17.07.2019. New York: ACM, pp. 985-993.
http://doi.acm.org/10.1145/3321707.3321841

gives Gin a large set of potential contributors and users. Further, Java also allows Gin to leverage powerful off-the-shelf tooling, such as JavaParser [11], JUnit [12] and Surefire [13].

Gin is necessarily *both* extensible and modifiable because it must not constrain scientific inquiry into GI. Thus, Gin is a *toolbox*, rather than a framework, which is only extensible. GI aspires to automate code improvement tasks. This goal, coupled with GI's open problems, has a number of immediate consequences for Gin's design: Gin must build and scale to industrial code and it must smoothly and easily support adding new search strategies, sampling strategies, and mutation operators.

To smoothly run tests on program variants, Gin understands the two currently dominant Java build systems — Maven and Gradle; as outlined in Section 2.6 it builds such projects automatically, obviating shell commands. To scale, Gin utilises dynamic compilation, which recompiles only changed classes and their dependencies, and online classloading. These features allow it to modify, recompile, and execute large-scale systems within a single virtual machine (see Section 2.3).

Out of the box, Gin supports an array of program transformations and two representations of code — ASTs and token streams. Both representations are extremely flexible and support operations at multiple granularities (subtrees or grammatical units); in contrast with other approaches, Gin presents the raw representation, without filtering, to a mutation operator. These unfiltered representations free researchers to define custom operators (see Section 3.1), like one that considers comments. Gin's design carefully separates search from applying transformations and evaluating fitness, which involves building and testing a variant. As a consequence of this separation, one need only specify a sequence of transformations to define a new search strategy. Gin also provides a sampling feature that reports the test case results for the single application of any operator (Section 3.3). With the notable exception of De Souza et al. [14], which uses dynamic analysis to consider intermediate execution state, all GI work thus far assumes Boolean test cases in their fitness evaluation. Gin is the first to record the expected and actual results, allowing researchers to define more fine-grained fitness functions and smooth the search space landscape.

In addition to its feature set for general GI, Gin is the first to support two innovative features for NFI — built-in profiling and automated test generation. A key to scaling GI is narrowing its search to code fragments. APR has successfully used fault localisation for this purpose. In NFI, the natural analog is profiling. Thus, profiling [39] is integral to Gin, freeing researchers to narrow the search for nonfunctional improvements to, for instance, the most time-consuming methods. GI usually relies on testing to measure the fitness of evolved software variants [32]. In NFI, one can always use the original program as a test oracle and use it to automate test case generation. Gin leverages this insight to be the first GI tool to incorporate an automated test case generation tool (EvoSuite [15]).

A university course on GI has already been delivered using the first release version of Gin, demonstrating Gin's ease of use and flexibility (Section 3.6). This paper makes two principle contributions: the design and architecture of Gin and its instantiation in Java for the Java ecosystem. Gin is open source and available online http://github.com/gintool/gin/.

2 ARCHITECTURE

Figure 1 presents a high-level overview of Gin's two main pipelines, and a UML class diagram of the core classes is given in Figure 2. Gin's core functionality is divided between the manipulation of source code, and unit test execution. Tools that can be used independently of source editing and evaluation, such as test generation and profiling, are grouped together in the `gin.util` package and omitted from the class diagram.

The pipelines in Figure 1 give two example uses of Gin: preprocessing to identify code of interest within a project, and search space analysis. A complete profiling pipeline is provided by the `gin.util.Profiler` class, which will output 'hot methods' as suitable targets for improvement.

The analysis of GI search spaces is of increasing research interest [8, 26, 35] and Gin facilitates this process: the toolkit includes several examples of search space tools that sample and enumerate the space of program edits. Adding new edit types and reusing this code is straightforward. Gin will sample the patch space, running the specified tests against each patch and record the result: whether the patch is valid, the result of compilation, the test output, run times, and error details. Test suites can be generated in any manner for use in Gin, provided they are in JUnit format. Most previous GI work only considered Boolean test case results during fitness evaluation; by recording more detailed test output, Gin supports the implementation of more fine-grained fitness functions.

A major use case of Gin is to apply GI to improve code: Gin deliberately delegates the design of search algorithms to the user, but a simple example of a local search algorithm is included. As the code examples in Section 3 shows, it is straightforward to incorporate Gin features into other search algorithms and applications.

2.1 Patch-Edit Model and Representation

Following standard practice in GI [28], the basic representation used by Gin is a patch to be applied to the source code. Each patch is a list of edits, and each edit is the application of a single operator to the target source code.

The original source code is loaded into a `SourceFile` object. There are two subclasses of `SourceFile`: `SourceFileLine` focused on line-level edits and `SourceFileTree` focused on edits to the Abstract Syntax Tree (AST). Each line in the source file and each node in the AST is allocated a unique ID; these IDs are referenced by edits, simplifying the problem of resolving patches containing multiple edits to the same location(s). For example, if an edit applies to a particular ID, but that ID no longer exists due to a previous delete, the edit will gracefully degrade to a no-op.

`SourceFile` is immutable: any methods that modify the source return a modified copy rather than changing the internal state of the `SourceFile`. Thus a patch is a sequence of edits, each producing a new `SourceFile`, which simplifies the implementation of new edits: an edit must simply accept a `SourceFile` and return a new one with the edit applied.

Gin includes subclasses of the `Edit` class that implement line and node operators commonly used in the literature, and examples of more fine-grained operators that replicate operators commonly found in the mutation testing domain (see Section 2.2). Edits may be targeted to specific locations. For example, a *copy* may copy a

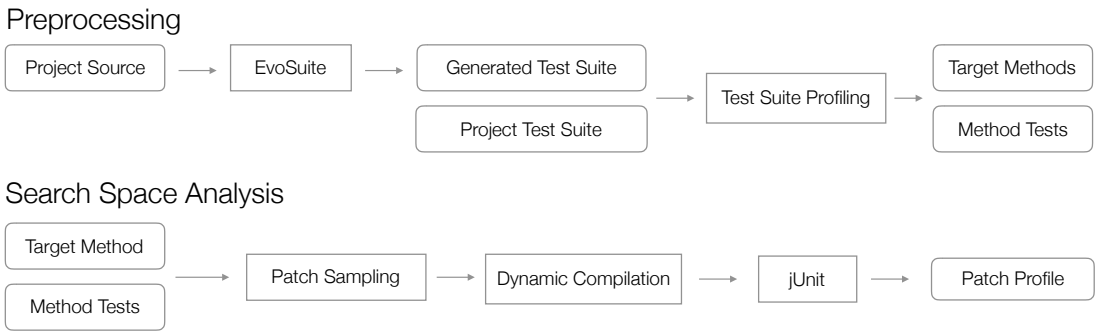

Figure 1: Gin Pipelines

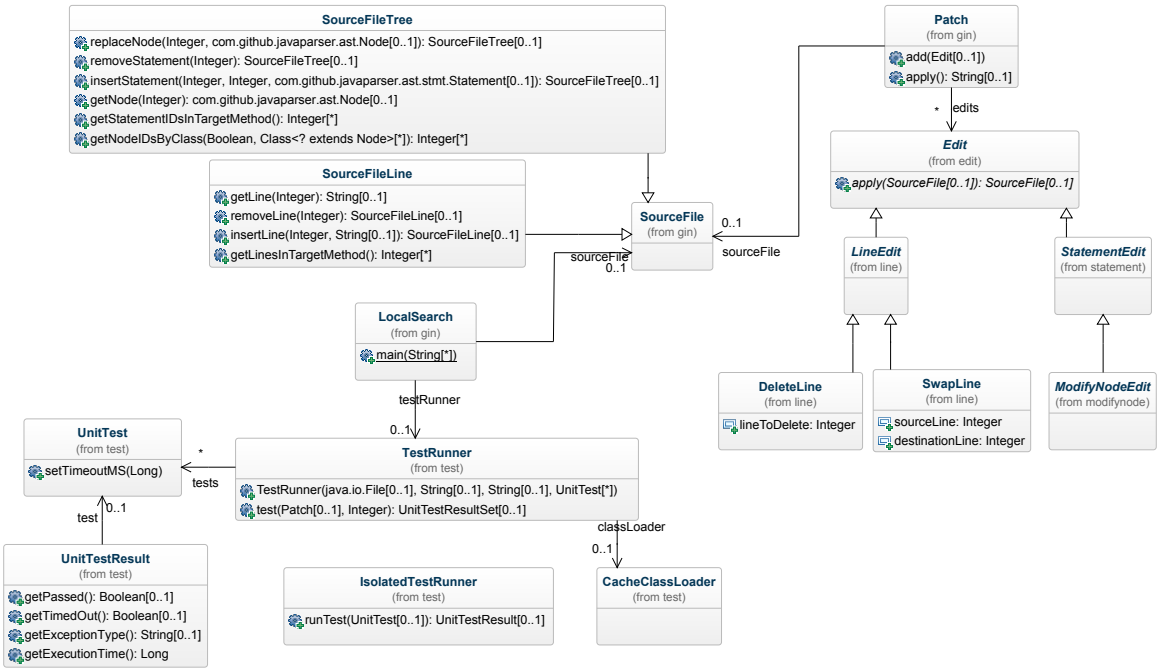

Figure 2: Gin core classes. Attributes are omitted and only a subset of method signatures are shown for simplicity. Also note that several Edit implementations and supporting utility classes are omitted.

statement or line from anywhere in the source, but limit its target location to locations of a certain type, or within a given method. `SourceFile` can be instantiated with a target method or methods, and will then provide a list of locations limited to those methods.

`SourceFile` provides methods for manipulating source:

Accessors return lists of IDs corresponding to a given language construct, e.g. an **`if`** statement or all block statements.
Getters return a copy of a line or AST node specified by an ID
Setters update the source file by deleting, inserting, or replacing at a specified location.
Convenience methods perform common tasks, for example selection of a random statement.

`SourceFile` also provides methods to generate the modified Java source for compilation and execution.

2.2 Operator Sets

Gin currently implements four sets of `Edit` operators:

(1) Line edits: Delete, Replace, Copy, Swap, Move.
(2) Statement edits: Delete, Replace, Copy, Swap, Move.
(3) Constrained (Matched) statement edits: Replace, Swap.

(4) Micro edits: BinaryOperatorReplacement.

The first two represent canonical transformations from the GI program optimisation and program repair literature respectively. The line edits can be found in the work of Petke and Langdon, particular in the GISMOE tool [24, 25, 33, 34]. The statement edits were first used in the seminal GenProg [27] automated program repair tool, and the others are proposed in this paper.

Constrained edits limit the canonical transformations to compatibility within the Java grammar: for example, swapping a 'do statement' with another 'do statement'; the intuition behind such operators is that they are more likely to make replace and swap operations between related program sites, and are less likely to lead to program disruption. A more refined analogue to constrained edits can be found in ARJA [42], a Java APR tool, which limits replacements to program elements that are both structurally and type-compatible. The fourth type are similar to the micro-mutations of [18], and numerous examples in the mutation testing literature (such as [30]). For example, binary operators replacement will consider replacing == with !=.

Providing implementations of all these operators within one toolkit simplifies experimental comparisons and analysis. Gin is designed so that adding new operators is simple: an example of one of the existing implementations is given Section 3.

2.3 Dynamic Class Loading and Test Execution

Once source code has been edited, it must be evaluated. Gin invokes test cases using JUnit and provides the information needed to target functional objectives and run-time performance. It reports the wall clock and CPU time of test execution over multiple measures, and returns details of the unit test outcome: whether the test passed, the expected and actual results, and details of any encountered errors, such as exceptions.

Compilation and test execution is performed entirely within memory to improve performance: there are no external command invocations and no JVMs are created. To achieve this, Gin uses a custom fork of the InMemoryCompilation project [40] to generate bytecode for the modified class, before loading the class in a custom ClassLoader that "overlays" the existing class hierarchy so that JUnit loads the modified class. This dynamic loading supports both individual source files and files contained within a larger project.

This complexity is hidden from the user, who instantiates and invokes the `TestRunner` with a patch, a reference to the original source file, and a list of unit tests. A collection of `UnitTestResult` objects is then returned indicating the outcome of the tests and the execution time. The existing utility classes for sampling and local search demonstrate how this can be done in practice with just a few lines of code (examples in Section 3).

2.4 Test Suite Generation

Test suites play a critical role in determining the outcome of GI [38, 41]. By standardising on JUnit for testing, Gin can exploit the unit test suite provided with a project; such suites usually provide good coverage, and are used by developers to test realistic use-cases for the code. In addition, automated test suite generation is provided via integration with the EvoSuite [2] tool. In the case of NFI, this test generation can be used to produce an independent oracle.

In order to facilitate experimentation, we have preconfigured EvoSuite to produce deterministic results. Moreover, the implemented `TestCaseGenerator` works out-of-the-box for Maven projects, modifying the pom file automatically to add necessary dependencies and modifying the output directory for Maven's test task. Semi-automated test case generation is supported for Gradle.

2.5 Profiling

The search space for software transformation is vast [26], and restricting the subspace explored by any improvement or repair algorithm is therefore critical in reducing search run-time. One of the main innovations of the GenProg repair tool [27] was to use fault localisation to reduce the size of the search space. Similarly, Gin provides a profiling capability to identify those parts of the software most exercised by the project's unit tests; we make the assumption that the provided unit tests are representative of real-world use, or at least they exercise the code where improvement is to be targeted. As Gin accepts a JUnit test suite as input, it is straightforward for a developer to provide a test suite that can guide Gin's improvements. For example, if a particular part of a project is known to be problematic, a small test suite can be provided to Gin that includes tests extensively targeting the problematic code surface. If reducing execution time is the goal, this may simply require providing the existing performance tests that many projects include.

As detailed in Section 2.6, Gin will automatically integrate with popular Java build tools, and the profiler `gin.util.Profiler` uses this facility to invoke and profile a project's unit tests. First, Gin invokes the entire test suite via the build tool's API, and parses the test reports to produce a list of tests, their containing classes, parameters, and whether they passed or failed. It then profiles individual tests by invoking them via the build tool API whilst enabling CPU sampling via the hprof profiler. The hprof profiler is somewhat dated, but it is sufficient for most projects; it is included in the Java 8 SDK that Gin requires, and at run-time provides a sample of the call stack every 10ms, which enables Gin to provide a list of frequently used methods. We use hprof as opposed to VisualVM and other alternatives due to its simplicity and batch operation: VisualVM is an interactive tool but Gin's profiling is automated; alternative profilers either are similarly interactive or not freely available.

The profiles are parsed by Gin and combined into a simple CSV file for use by researchers or later stages of Gin's pipeline; this component is standalone and can be used for projects outside of GI. For each method, a count giving the number of times the method is seen at the top of the call stack is provided, along with a list of all unit tests where the method was seen at the top of the call stack during profiling. In order to provide the list of calling unit tests, Gin profiles each method individually rather than the whole suite: this is a time-intensive process that can take many hours for very large projects, but need only be run a single time. A sample of a project's unit tests may be requested instead. The profiler is separate from the core of Gin and therefore easily bypassed by researchers who do not require it.

2.6 Build Tool Integration

One of the goals of Gin is to enable systematic experimentation on real-world code; this requires the ability to compile, package and test a diverse set of large projects. Fortunately, the Java ecosystem has converged to a small number of build tools that support these requirements and provide functionality through APIs. In particular, the Gradle [3] and Maven [1] build tools are very popular and used by over 95% of developers responding to one recent survey [4]; Gradle is the default build tool of the Android ecosystem, and almost all the GitHub projects we have examined during empirical work with Gin use one of the two tools. This standardisation enables Gin to accept most Java projects without modification, and run tasks without resorting to simply invoking shell commands.

Despite their popularity, the documentation of the APIs for both Gradle and Maven is somewhat sparse, and requires a certain amount of experimentation and reverse engineering; most of what we learnt during the process has subsequently been captured in the `gin.util.Project` class, which can be used outside of Gin to examine and manipulate projects, lowering the overhead for other researchers. For example, the `Project` class will provide the classpath for a project, find a particular source file within a project's file hierarchy, provide a standard method signature for a given method, provide a list of project tests, or run a unit test given its name. The `Project` class is used by the `Profiler` and other parts of Gin to interrogate and manipulate a project, and thus support for a new build tool can be added by modifying just this class.

Gin can infer the necessary classpath and dependencies for running unit tests from a Maven or Gradle project, or these can be specified manually.

3 IN PRACTICE

We now demonstrate the simplicity and extensibility of Gin with code examples for common use-cases.

3.1 Implementing New Edits

Whilst Gin contains canonical edit operators from the literature and some novel operators, development of such operators remains an area of active research; implementation of new edit types in Gin is therefore made as simple as possible. Code for a `ReplaceStatement` is given in Listing 1. An edit must provide:

- a constructor returning a random instance of the edit; we use methods in `SourceFile` to select two random statement IDs. The boolean argument to `getRandomStatementID` specifies whether the ID should be within the target method
- an `apply()` method to apply the edit on a given `SourceFile`. Here, the method replaces the statement at destinationID with a clone of the statement at sourceID.

Listing 2 shows an implementation of the matched equivalent of a replace statement edit. This extends the existing `ReplaceStatement` edit, constraining the source statement to be of the same type as the destination statement.

```
1 public class ReplaceStatement extends StatementEdit {
2
3   public int sourceID;
4   public int destinationID;
5
6   public ReplaceStatement(SourceFileTree sf, Random r) {
7     sourceID = sf.getRandomStatementID(false, r);
8     destinationID = sf.getRandomStatementID(true, r);
9   }
10
11  public SourceFile apply(SourceFileTree sf) {
12    Statement source = sf.getStatement(sourceID);
13    Statement dest = sf.getStatement(destinationID);
14    return sf.replaceNode(dest, source.clone());
15  }
16
17 }
```

Listing 1: Implementing an edit in Gin

```
1 public class MatchedReplaceStatement extends
      ReplaceStatement {
2   public MatchedReplaceStatement(SourceFileTree sf,
3                                  Random r) {
4     super(0, 0);
5     destinationID = sf.getRandomStatementID(true, r);
6     sourceID = sf.getRandomNodeID(false,
7     sf.getStatement(destinationID).getClass(), r);
8   }
9 }
```

Listing 2: Extending an existing edit in Gin

3.2 A Simple Search Algorithm

A condensed version of the local search example provided in Gin is given in Listing 3. The search starts with a single-edit random patch and at each step a random edit is removed or a new randomly-generated edit is added. If the new patch offers an improvement, it is retained and the process repeated. The only important code not shown is the instantiation of `SourceFile` with a source file and target method, and `TestRunner` with a working directory, classpath and list of tests. Local search can also be invoked from the terminal, assuming the test class `ExampleTest`:

```
java gin.LocalSearch examples/Example.java
```

Additional arguments allow the user to specify more unit tests, a classpath, target methods, operators and so on. The search can also be invoked programmatically: a call to `localSearch = new LocalSearch("examples/Example.java")` will create the local search object for the specified target source file, and then `Patch result = localSearch.search();` will run the search, returning a reference to a `Patch` object with the best patch found.

Extension of this search algorithm to a population-based evolutionary algorithm is simple. The only additions required are selection (which can use the existing time and unit test methods to rank solutions) and a concept of crossover, which can be performed at the Patch level, recombining different combinations of edits.

3.3 Sampling and Enumeration

Essential to search space analysis is the ability to systematically generate variants of the original program code. Gin gives examples

```
1 private Patch search() {
2     // start with the empty patch
3     Patch bestPatch = new Patch(sourceFile);
4     long bestTime = testRunner.test(bestPatch, 10).
          totalExecutionTime();
5
6     for (int step = 1; step <= NUM_STEPS; step++) {
7         Patch neighbour = neighbour(bestPatch, rng);
8         UnitTestResultSet rs = testRunner.test(neighbour
              ,10);
9         if (rs.getValidPatch() && rs.getCleanCompile() &&
10            rs.allTestsSuccessful() &&
11            rs.totalExecutionTime() < bestTime) {
12          bestPatch = neighbour;
13          bestTime = rs.totalExecutionTime();
14        }
15    }
16
17    return bestPatch;
18 }
19
20 public Patch neighbour(Patch patch, Random rng) {
21    Patch neighbour = patch.clone();
22
23    if (neighbour.size() > 0 && rng.nextFloat() > 0.5) {
24        neighbour.remove(rng.nextInt(neighbour.size()));
25    } else {
26        neighbour.addRandomEdit(rng, allowableEditTypes);
27    }
28
29    return neighbour;
30 }
```

Listing 3: Local search in Gin

for sampling and enumerating the search space and writing results to a comma-separated file (e.g. Figure 3): the intention is that these can easily be modified or extended to suit experimental needs. The user only needs to provide method names and associated unit tests in a file, which could simply be the Profiler's output file. We provide a helper abstract Sampler class for sampling and enumerating edits, as well as three sub-classes:

EmptyPatchTester will run all unit tests through Gin, and save results to a file.

RandomSampler will make a number of random edits, test the resulting source, and return the result.

DeleteSampler will enumerate all possible `DeleteLine` and `DeleteStatement` edits for a method, test the resulting source, and save results to a file.

3.4 Profiling and Generating Tests

Test suites can be created and a profiler invoked with single terminal commands. To generate new test cases:

```
java -cp build/gin.jar gin.util.TestCaseGenerator
-projectDir examples/maven-simple -projectName my-app
-classNames com.mycompany.app.App -generateTests
```

To profile a test suite:

```
java -cp build/gin.jar gin.util.Profiler -p my-app
-d examples/maven-simple/ .
```

Results are written to `profiler_output.csv`.

3.5 Implementing an Enumerator

Consider an enumerator to exhaustively apply an edit at every possible location in a code region, perhaps to perform landscape analysis. Taking the example of `DeleteEdit`, Listing 4 gives the requisite source code.

The code here accepts a single class example program, but could be extended to large projects with a few lines specifying the working directory, classpath etc. In the example, we specify an array of UnitTests to be applied to the modified code. We set a number of repeats for each test. We then create `SourceFileTree` and `TestRunner` objects to perform the analysis. We create an empty patch and test that as a baseline. Finally, we get a list of all statement IDs in the source, and enter a loop that creates and tests a `DeleteStatement` for each statement. The results are written to file by an auxiliary method.

3.6 Case Study - An Application in Teaching

The ease with which Gin can be deployed and modified has been demonstrated by its use in teaching. In 2017 and 2018 two of the authors used the first release version of Gin as a vehicle to teach concepts in GI to two moderately sized classes of students (26 and 51 students respectively) in a fourth year Search Based Software Engineering course. In each class a group assignment[2] required students to:

(1) Download, build and run Gin;
(2) Run Gin using the LocalSearch method to improve the runtime of four example programs;
(3) Write a qualitative and quantative analysis describing the type and distributions of patches in the best-performing programs;
(4) Extend Gin to minimise the length of the best patches; and
(5) Apply Gin to their own benchmark program and analyse the results.

Each group produced a report outlining the findings from steps 2-4. There were 12 group reports submitted for the first cohort and 13 group reports for the second. All groups were able to quickly deploy Gin, run the four benchmarks in step 2 and reliably produce better variants of the example programs. In step 3 students were required to modify the Gin implementation. Students used a variety of approaches, ranging from brute-force enumeration of edit subsets to greedy algorithms through to search heuristics such as A*.

Students were able to modify Gin with some ease, with some groups simply extending the local search example code while others went so far as to implement patch minimisation. The extended implementations were able to verify both the preservation of code structure and application performance.

In step 4 groups used a variety of benchmarks and showed an awareness of code features that were amenable to the set of GI operators used in this assignment. Students sought out examples that

[2]The assignment material is publicly available at https://github.com/anon-sbse-teacher/project and https://github.com/markuswagnergithub/SBSEcourse/.

PatchIndex	PatchSize	Patch
1	1	| gin.edit.statement.SwapStatement ./src/main/java/org/jcodec/codecs/vpx/VPXBitstream.java:752 <-> ./src/main/java/org/jcodec/codecs/vpx/VPXBitstream.java:884 |
2	1	| gin.edit.statement.ReplaceStatement ./src/main/java/org/jcodec/codecs/prores/ProresEncoder.java:2310 -> ./src/main/java/org/jcodec/codecs/prores/ProresEncoder.java:1185 |
3	1	| gin.edit.statement.CopyStatement ./src/main/java/org/jcodec/containers/mp4/boxes/Box.java:514 -> ./src/main/java/org/jcodec/containers/mp4/boxes/Box.java:110:110 |

MethodIndex	TestIndex	UnitTest	RepNumber	PatchValid	PatchCompiled	TestPassed	TestExecutionTime(ns)	TestCPUTime(ns)
152	1	org.jcodec.codecs.vpx.TestCoeffEncoder.testCoeffDCTU []	0	TRUE	TRUE	FALSE	2853708	1535633
189	1	org.jcodec.codecs.prores.ProresEncoderTest.testWholeThing []	0	TRUE	FALSE	FALSE	0	0
184	1	org.jcodec.containers.mp4.boxes.TrunBoxTest.testReadWriteCreate []	0	TRUE	FALSE	FALSE	0	0

TestTimedOut	TestExceptionType	TestExceptionMessage	AssertionExpectedValue	AssertionActualValue
FALSE	java.lang.AssertionError	expected:<255> but was:<207>	255	207
FALSE	N/A	N/A	N/A	N/A
FALSE	N/A	N/A	N/A	N/A

Figure 3: Example output from a sampling run, split into three rows to save space

```
1 public static void main(String[] args) {
2
3   UnitTest[] ut = {
4     new UnitTest("TriangleTest","testInvalidTriangles"),
5     ...
6   };
7
8   int reps = 1;
9
10  SourceFileTree sf = new SourceFileTree(
11      "examples/simple/Triangle.java",
12      Collections.singletonList(
13           "classifyTriangle(int,int,int)"));
14
15  TestRunner tr = new TestRunner(
16      new File("examples/simple"), "Triangle",
17      "examples/simple", Arrays.asList(ut));
18
19  // Start with the empty patch
20  Patch patch = new Patch(sf);
21
22  // Run empty patch and log
23  UnitTestResultSet rs = tr.test(patch, reps);
24  writeResults(rs, 0);
25
26  int patchCount = 0;
27  for (int id : sf.getStatementIDsInTargetMethod()) {
28    patchCount++;
29    patch = new Patch(sf);
30    patch.add(new DeleteStatement(sf.getFilename(),id));
31
32    rs = tr.test(patch, reps);
33    writeResults(rs, patchCount);
34  }
35 }
```

Listing 4: Implementing a delete enumerator. This is the complete code excepting some straightforward processing in writeResults() to write out the results to a CSV file.

were amenable to optimisations such as invariant hoisting and removal of redundant code. Some submissions also demonstrated the effectiveness of Gin in improving program performance when potentially useful raw materials (such as redundant conditionals) are introduced into code. In summary, Gin serves well in an educational setting because it presents so few barriers to experimentation.

4 RELATED WORK

Genetic improvement tools can be divided into two categories: those that focus on improvement of functional (FI), and non-functional software properties (NFI). The tools in the first category mainly come from the field of automated program repair (APR). The canonical example of these is GenProg [27], the first GI tool scaling to large real-world instances.

Early work in APR focused on fixing C and C++ programs; only more recently have other languages, such as Java, been considered. For example, Martinez and Monperrus released ASTOR, a program repair library for Java that implements several program repair approaches [31]. It allows for the addition of new tools, but does not facilitate more fine-grained extension, such as the addition of a single mutation operator or search strategy.

Genetic improvement has also been used to add new features to software. Such work has also mostly focused on C code; for example, the FI tool used by Barr et al. [6] in their automated software transplantation work is available online.

Several NFI frameworks have been developed, though few have been made open source. One of the largest is by Langdon et al. [21–24], and focuses on runtime improvement of C and C++ programs. Depending on the particular variant of their framework, line-level or expression-level changes are possible. The locoGP [10] framework developed by Cody-Kenny et al. evolves entire Java AST's and acts as an off-the-shelf optimisation tool, while allowing limited customisation via its fitness function.

Several attempts have been made to provide a more extensible set of tools for optimising software using GI. GrammaTech's software evolution library (SEL) [36] enables the programmatic modification and evaluation of extant software. Its API defines software objects using the Common Lisp Object System (CLOS) to provide a uniform interface, allowing it to manipulate many software artifacts, ranging from C source code, compiled assembler to limited support for Java. It also allows for addition of new mutation operators. However, modifying the framework presents a steep learning curve, particularly for those not familiar with Lisp. A Genetic Programming microframework, MicroGP, has been used as a basis for a language-independent GI framework, but the source code is no longer available. Perhaps the work mostly closely sharing the goals of Gin is the Python GI framework PyGGI [16].

5 CONCLUSION

GI is a maturing research topic, with multiple examples of real-world deployment and a growing diversity of methods. We believe shared tooling is essential to further advance in this area. As such, we have described Gin, a platform for GI experimentation in Java.

Gin offers great extensibility, yet remains simple to use. It integrates with the industry-standard Gradle and Maven build tools, allowing experimentation with real-world software projects. Gin also integrates with established tools such as EvoSuite to automatically generate unit tests, as well as SureFire and hprof for profiling. As a further contribution, we have captured the experience our team have gained in integrating with Gradle and Maven builds in `gin.util.Project` class, which can also be used in isolation for researchers interested in other aspects of software experimentation.

We now call for participation: researchers are encouraged to download the tool from http://github.com/gintool/gin/, and experiment with the example programs we have included. Anyone working in GI is also encouraged to report bugs, raise feature requests, and contribute documentation, examples and additional features to the platform via our GitHub project.

Immediate plans for future development of the platform include more seamless automated integration of generated tests with Gradle; further edit operators, search methods and objective functions; more landscape sampling and enumeration tools, and additional use-case scenarios.

Data Access Statement. The source code of Gin can be obtained from https://github.com/gintool/gin.

ACKNOWLEDGMENTS

The work in this paper was part funded by the UK EPSRC [grants EP/J017515/1 and EP/P023991/1]; and Australian Research Council Project DE160100850. We are also grateful to the developers of JavaParser for promptly fixing a bug discovered during Gin's development.

REFERENCES

[1] [n. d.]. Apache Maven Project. https://maven.apache.org. ([n. d.]). Accessed: 2019-01-20.

[2] [n. d.]. EvoSuite: Automatic Test Suite Generation for Java. ([n. d.]). http://www.evosuite.org/

[3] [n. d.]. Gradle Build Tool. https://gradle.org. ([n. d.]). Accessed: 2019-01-20.

[4] [n. d.]. The State of Java in 2018. https://www.baeldung.com/java-in-2018. ([n. d.]). Accessed: 2019-01-20.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

[6] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, Michal Young and Tao Xie (Eds.). ACM, 257–269. https://doi.org/10.1145/2771783.2771796

[7] James E Bessen, Stephen Michael Impink, Robert Seamans, and Lydia Reichensperger. 2018. The Business of AI Startups. *Boston Univ. School of Law, Law and Economics Research Paper* 18-28 (2018).

[8] B. R. Bruce, J. Petke, M. Harman, and E. T. Barr. 2018. Approximate Oracles and Synergy in Software Energy Search Spaces. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2827066

[9] Doug Burger and Todd M Austin. 1997. The SimpleScalar tool set, version 2.0. *ACM SIGARCH computer architecture news* 25, 3 (1997), 13–25.

[10] Brendan Cody-Kenny, Edgar Galván López, and Stephen Barrett. 2015. locoGP: Improving Performance by Genetic Programming Java Source Code. In *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, Sara Silva and Anna Isabel Esparcia-Alcázar (Eds.). ACM, 811–818. https://doi.org/10.1145/2739482.2768419

[11] JavaParser Community. 2019. JavaParser — For Processing Java Code. http://javaparser.org. (2019). [Online; accessed 6-February-2019].

[12] JUnit Community. 2019. JUnit Webpage. https://junit.org/junit5/. (2019). [Online; accessed 6-February-2019].

[13] Maven Surefire Community. 2019. Maven Surefire Plugin. https://maven.apache.org/surefire/maven-surefire-plugin/. (2019). [Online; accessed 6-February-2019].

[14] Eduardo Faria de Souza, Claire Le Goues, and Celso Gonçalves Camilo-Junior. 2018. A Novel Fitness Function for Automated Program Repair Based on Source Code Checkpoints. (2018).

[15] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 416–419.

[16] Jinhan Kim Gabin An and Shin Yoo. 2018. Comparing Line and AST Granularity Level for Program Repair using PyGGI. In *Genetic Improvement Workshop, co-located with ICSE 2018*.

[17] Daniel Gracia Perez, Gilles Mouchard, and Olivier Temam. 2004. MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms.

[18] Saemundur O Haraldsson, John R Woodward, Alexander EI Brownlee, and David Cairns. 2017. Exploring Fitness and Edit Distance of Mutated Python Programs. In *European Conference on Genetic Programming*. Springer, 19–34.

[19] Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, and Kristin Siggeirsdottir. 2017. Fixing bugs in your sleep: how genetic improvement became an overnight success. In *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, Peter A. N. Bosman (Ed.). ACM, 1513–1520. https://doi.org/10.1145/3067695.3082517

[20] Yue Jia, Ke Mao, and Mark Harman. 2018. Finding and fixing software bugs automatically with SapFix and Sapienz. https://code.fb.com/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/. (2018). [Online; accessed 6-February-2019].

[21] William B. Langdon and Mark Harman. 2010. Evolving a CUDA kernel from an nVidia template. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010*. IEEE, 1–8. https://doi.org/10.1109/CEC.2010.5585922

[22] William B. Langdon and Mark Harman. 2015. Optimizing Existing Software With Genetic Programming. *IEEE Trans. Evolutionary Computation* 19, 1 (2015), 118–135. https://doi.org/10.1109/TEVC.2013.2281544

[23] William B. Langdon, Mark Harman, and Yue Jia. 2010. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software* 83, 12 (2010), 2416–2430. https://doi.org/10.1016/j.jss.2010.07.027

[24] William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. 2015. Improving CUDA DNA Analysis Software with Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*, Sara Silva and Anna Isabel Esparcia-Alcázar (Eds.). ACM, 1063–1070. https://doi.org/10.1145/2739480.2754652

[25] William B. Langdon and Justyna Petke. 2017. Software is not fragile. In *First Complex Systems Digital Campus World E-Conference 2015*. Springer, 203–211.

[26] William B. Langdon, Nadarajen Veerapen, and Gabriela Ochoa. 2017. Visualising the Search Landscape of the Triangle Program. In *Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings (Lecture Notes in Computer Science)*, James McDermott, Mauro Castelli, Lukás Sekanina, Evert Haasdijk, and Pablo García-Sánchez (Eds.), Vol. 10196. 96–113. https://doi.org/10.1007/978-3-319-55696-3$_7$

[27] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[28] Claire Le Goues, Westley Weimer, and Stephanie Forrest. 2012. Representations and operators for improving evolutionary software repair. In *Genetic and Evolutionary Computation Conference, GECCO '12, Philadelphia, PA, USA, July 7-11, 2012*, Terence Soule and Jason H. Moore (Eds.). ACM, 959–966. https://doi.org/10.1145/2330163.2330296

[29] Sean Luke. 2017. ECJ then and now. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, 1223–1230.

[30] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability* 15, 2 (2005), 97–133.

[31] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 441–444. https://doi.org/10.1145/2931037.2948705

[32] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, David R. White, Woodward, and John R. Woodward. 2017. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* (2017). https://doi.org/10.1109/TEVC.2017.2693219

[33] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2014. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *Genetic Programming - 17th European Conference, EuroGP 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers (Lecture Notes in Computer Science)*, Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo García-Sánchez, Juan J. Merelo, Víctor Manuel Rivas Santos, and Kevin Sim (Eds.), Vol. 8599. Springer, 137–149. https://doi.org/10.1007/978-3-662-44303-3$_1$2

[34] Justyna Petke, William B Langdon, and Mark Harman. 2013. Applying genetic improvement to MiniSAT. In *International Symposium on Search Based Software Engineering*. Springer, 257–262.

[35] Joseph Renzullo, Stephanie Forrest, Westley Weimer, and Melanie Moses. 2018. Neutrality and Epistasis in Program Space. In *Genetic Improvement Workshop, co-located with ICSE 2018.*

[36] Eric Schulte. 2014. *Neutral Networks of Real-World Programs and their Application to Automated Software Evolution.* Ph.D. Dissertation. University of New Mexico, Albuquerque, USA. https://cs.unm.edu/ eschulte/dissertation.

[37] Eric M. Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software mutational robustness. *Genetic Programming and Evolvable Machines* 15, 3 (2014), 281–312. https://doi.org/10.1007/s10710-013-9195-8

[38] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 532–543. https://doi.org/10.1145/2786805.2786825

[39] Oracle Systems. 2019. HPROF: A Heap/CPU Profiling Tool. https://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html. (2019).

[40] Nguyen Kien Trung. 2019. In Memory Java Compiler. https://github.com/trung/InMemoryJavaCompiler. (2019).

[41] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18.* ACM Press. https://doi.org/10.1145/3180155.3180182

[42] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering* (2018).