Comparing Line and AST Granularity Level for Program Repair using PyGGI

Gabin An
KAIST
Daejeon, Republic of Korea
agb94@kaist.ac.kr

Jinhan Kim
KAIST
Daejeon, Republic of Korea
jinhankim@kaist.ac.kr

Shin Yoo
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

ABSTRACT

PyGGI is a lightweight Python framework that can be used to implement generic Genetic Improvement algorithms at the API level. The original version of PyGGI only provided lexical modifications, i.e., modifications of the source code at the physical line granularity level. This paper introduces new extensions to PyGGI that enables syntactic modifications for Python code, i.e., modifications that operates at the AST granularity level. Taking advantage of the new extensions, we also present a case study that compares the lexical and syntactic search granularity level for automated program repair, using ten seeded faults in a real world open source Python project. The results show that search landscapes at the AST granularity level are more effective (i.e. eventually more likely to produce plausible patches) due to the smaller sizes of ingredient spaces (i.e., the space from which we search for the material to build a patch), but may require longer time for search because the larger number of syntactically intact candidates leads to more fitness evaluations.

CCS CONCEPTS

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

ACM Reference Format:
Gabin An, Jinhan Kim, and Shin Yoo. 2018. Comparing Line and AST Granularity Level for Program Repair using PyGGI. In *GI'18: GI'18:IEEE/ACM 4th International Genetic Improvement Workshop , June 2, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194810.3194814

1 INTRODUCTION

Most of the currently available Genetic Improvement (GI) [17] tools are meant to be research prototypes, and naturally are either tied or specialised to particular research ideas. For example, `astor` provides Java re-implementations of a series of widely studied GI approaches [13]: while the re-implementations share common infrastructures provided by `astor` itself, one may argue that the aim of the framework is to provide a suite of GI approaches, rather than to provide a general framework at the API level. GIN is designed to be a general framework [21], but its current implementation has some limitations such as only being able to operate one Java source file. Additionally, GIN's implementation language, Java, can pose inherent overhead to fast prototyping. Consequently, in practice, it may be hard to use those implementations as a baseline to build new GI tools and experiments. Since the essential parts of GI, such as program preprocessing, code modifications, or patch management, can impose considerable implementation overhead, a general framework that is extensible and easy to use may serve researchers and practitioners alike.

We have previously introduced the initial version of PyGGI [1] as a general framework of GI written in Python[1]. Our aim was to implement a simple, lightweight yet extensible framework, that can be used by researchers and practitioners to implement GI techniques of various flavours, rather than to promote a specific algorithm for GI. The initial version only provided lexical modification, i.e., insertion, copy, and replacement of physical lines of program source code. The lexical modifications had the benefit of being language agnostic, as found in other techniques such as Observational Slicing [3, 7].

In this paper, we introduce the extended version of PyGGI that supports syntactic modifications for Python (i.e., modifications that operate on parsed Python code) as well as fault localisation supports. In addition, exploiting the new extensions, we also present an empirical study that compares the lexical and the syntactic search spaces for automated program repair using seeded faults in an open source project. The lexical search space is the space of all programs that can be constructed by application of lexical operators (i.e., copy, insertion, and replacement of physical lines), whereas the syntactic search space is the space of all programs that can be constructed by application of syntactic operators (i.e., modifications of abstract syntax tree nodes that correspond to Python statements).

The technical contributions of this paper is as follows:

- We present a new version of PyGGI which now supports syntactic modification at the statement level for Python, rather than the lexical modification at the physical line level in the previous version.
- We conduct a case study that compares line and AST granularity level and corresponding search landscapes for the statement level automated program repair (i.e., repairs that consist of copy, insertion, and replacement of program statements). We use a Python open source project, `sh`, and seeded faults for the evaluation. the results show that the AST level granularity and its search landscape are more effictive at producing plausible patches, as patches tend to lead to fewer syntactic errors.

We begin by describing the overall design of PyGGI.

GI'18, June 2, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5753-1/18/06…$15.00
https://doi.org/10.1145/3194810.3194814

[1]Available from https://github.com/coinse/pyggi.

2 DESIGN OF PYGGI

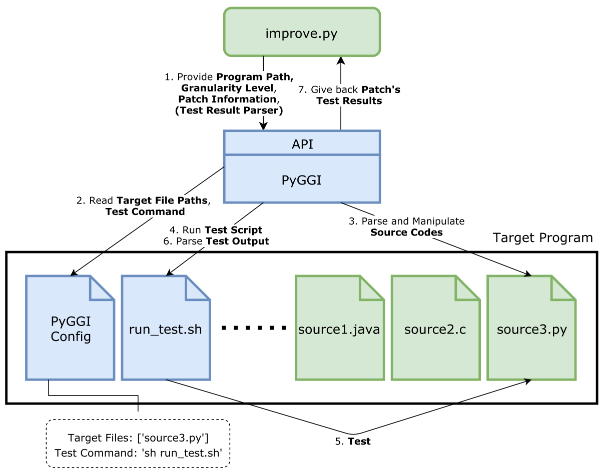
improve.py
1. Provide **Program Path, Granularity Level, Patch Information, (Test Result Parser)**
7. Give back **Patch's Test Results**
API
PyGGI
2. Read **Target File Paths, Test Command**
4. Run **Test Script**
6. Parse **Test Output**
3. Parse and Manipulate **Source Codes**
Target Program
PyGGI Config
run_test.sh
source1.java
source2.c
source3.py
Target Files: ['source3.py']
Test Command: 'sh run_test.sh'
5. **Test**

Figure 1: PyGGI Overview

Figure 1 illustrates the overall system of PyGGI version 1.1 and shows how it works through the example: green boxes represent either inputs user provides (i.e., source code to patch or test scripts to execute) or an GI program user writes using PyGGI API (e.g., `improve.py`). To run PyGGI, a user provides a target program and locates a configuration file in the root path of the program. The configuration file should contain the information about paths to the target source codes and a test command. With the inputs from the user, PyGGI preprocesses the source codes into own contents according to the given granularity level.

Depending on the algorithm, PyGGI modifies the target program with edit operators and applies them *lazily*; that is, PyGGI manages a sequence of edit operators and applies it only when the candidate patch needs to be evaluated. Generating a cloned temporary directory, PyGGI executes the test command to evaluate the candidate patch. The test program or script, i.e., `run_test.sh` in Figure 1, should yield output in a pre-defined format. Otherwise, the user should provide a test result parser. The whole process continues until the termination criteria are met.

2.1 PyGGI Classes

PyGGI is composed of several classes as shown in Figure 2. Compared to the initial version, `Edit` class have been replaced with two abstract classes, AtomicOperator and CustomOperator, to allow implementation of operators that correspond to different granularity levels. PyGGI also provides four child classes that actually instantiate operators that are specific to different granularity levels.

While it is not represented in Figure 2, PyGGI also contains a module named `algorithms`. It currently includes only one abstract class, `LocalSearch`, that implements the basic skeleton of a local search algorithm. The user can easily inherit and override this to implement multiple local search variants, such as the Hill climbing or Tabu search. PyGGI can also be intergrated with evolutionary computation libraries, such as DEAP [4], to take advantage of both single and multi-objective evolutionary algorithms.

2.1.1 Program. `Program` encapsulates the target program, especially its source code. The actual internal representation of the pogram depends on the choice of granularity level. See Section 2.2 for details.

2.1.2 Logger. `Logger` customises a logging object for a program instance to help recording various information during the GI process. It has file and stream handlers, and provides five logging levels: `debug`, `info`, `warning`, `error`, and `critical`.

2.1.3 GranularityLevel. `GranularityLevel` dictates multiple factors: how the program is pre-processed and internally stored, and which operators can be used during the search. It inherits Enum, a python built-in enumeration class, and currently has two members: `line` and `AST`.

2.1.4 Patch. `Patch` is a sequence of edit operators, both atomic operator and custom operator. During search iteration, PyGGI modifies the source code of the target program by applying a patch. To apply a patch, a sequence of edit operators should be translated into a sequence of atomic operators. It can be possible since a custom operator is essentially a list of atomic operators. Then, the atomic operators modify the source codes in order.

2.1.5 TestResult. `TestResult` stores the result of the test suite executions on the patched candidate program. The results contain whether compilation succeeded, the elapsed execution time, as well as any other user-defined test outputs.

2.1.6 AtomicOperator. `AtomicOperator` is an abstract class for PyGGI-provided operators. Currently, there are total four classes that inherit and specialise it as shown in Figure 2. See Section 2.3 for details of each.

2.1.7 CustomOperator. `CustomOperator` is an abstract class that provides a skeleton for a user-defined custom operator. Custom operators should be a sequence of atomic operators as mentioned in Section 2.1.4. When users implement their own operators, it must inherit the `CustomOperator` class and override some methods to define the intended behaviours. Custom operators can be conceptually described as a function, which takes some program elements as input and outputs a list of atomic operators that operate on them.

2.2 Program Preprocessing

PyGGI pre-processes the target program before manipulating its source code. There are currently two granularity levels, line and AST, which correspond to the lexical and the syntactic approaches, respectively. For the line granularity level, PyGGI transmutes source code into a list of code lines; for the AST granularity level, it parses the source code and stores the AST. The line granularity level offers an off-the-shelf, language agnostic modifications, whereas the AST granularity level provides more structured code modifications. In particular, AST level modifications produce patches that are more syntactically intact, because any modification of a statement AST node will include its subtree. Figure 3 illustrates this difference.

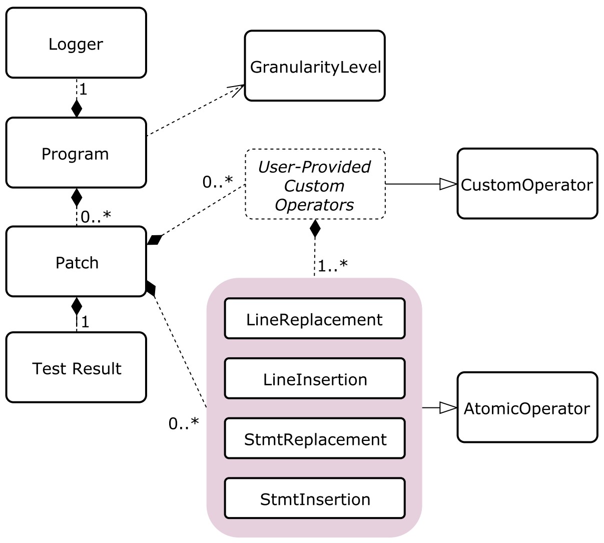

Figure 2: Simple Class Diagram of PyGGI

```
1 number = [-3, 4, -5, 6, 7]
2 pos_total = 0
3 for num in numbers:
4     if num > 0:
5         pos_total += num
6     print(num)
```

(a) Example Code

Line	Lexical	Syntactic
1	`number = [-3,4,-5,6,7]`	`number = [-3,4,-5,6,7]`
2	`pos_total = 0`	`pos_total = 0`

3	`for num in numbers:`	`for num in numbers:`
		`    if num > 0:`
		`        pos_total += num`

4	`    if num > 0:`	`if num > 0:`
		`    pos_total += num`

5	`        pos_total += num`	`pos_total += num`
6	`print(num)`	`print(num)`

(b) Modification Points

Figure 3: Comparision of modification points between lexical and syntactic approaches. Note that 1) lexical modification points contain indentation whitespace characters, and 2) syntactic statements include physical lines that correspond to its AST subtree.

2.3 Code Manipulation

PyGGI provides atomic operators that are based on the Plastic Surgery Hypothesis [2], which has been the fundamental assumption for other state-of-the-art repair techniques such as GenProg [8].

Table 1: The list of atomic operators provided by PyGGI, where the x and y are the indexes of modification points (note that *pos* is either *before* or *after*).

Gr.	Operator	Description
Line	LineReplacement(x, y)	Replace x with y (delete x if y is None)
	LineInsertion(x, y, pos)	Insert(copy) y before or after x
AST	StmtReplacement(x, y)	Replace x with y (delete x if y is None)
	StmtInsertion(x, y, pos)	Insert(copy) y before or after x

As mentioned in Section 2.1.6, PyGGI provides four atomic operators for manipulating the source codes of the program: LineReplacement and LineInsertion for the line level granularity, StmtReplacement and StmtInsertion for the statement level granularity. See descriptions of each operator in Table 1. The critical difference between Line and AST granularity level is the unit of modification. For example, if PyGGI attempts to insert the third *line* somewhere else, it would copy and insert only the single line that contains the `for` in Figure 3, with the risk of a syntax error. However, inserting the third *statement* means copying the entire loop with the body, thereby avoiding the potential syntax error.

Other custom operators can be generated by combining the atomic operators. For example, a deletion can be instantiated as a replacement with an empty line, whereas a move can be instantiated as an insertion followed up by a deletion, as shown in Table 2. Those operators in Table 2 are already implemented by PyGGI for both line and AST granularity level.

Another new feature in PyGGI allows the user to set modification weights for each modification point: this allows the user to focus modifications on specific parts of the source code. For example, the suspiciousness scores obtained by fault localisation techniques can be used as modification weights to make PyGGI focus on program elements that are more likely to be faulty. Once entered, the weights are normalised and used as the probability distribution of the roulette wheel selection of modification points by each modification operators. If no weight values are given, PyGGI uses an uniform distribution.

2.4 Validation & Evaluation

We need to apply the patch to the target program to evaluate a candidate program. To avoid any unwanted interference with the original program, PyGGI clones the entire original program into a temporary directory when a user requests applying the patch and makes edits in the directory.

After applying the patch, PyGGI executes the given test command to validate and evaluate the patch. We call a patch is *valid* if and only if it produces a syntactically correct program and the resulting test execution halts within the given time-out limit. The test results, which are printed out by the test execution, are parsed by either PyGGI or a user-provided result parser.

PyGGI returns the captured test results to the user so that they can make use of it for evaluating the patch. For example, test results can include the number of failing test cases or the information about memory consumption during the test execution; the user can write their own fitness function based on these results.

Table 2: The list of custom operators already implemented in PyGGI, where the x and y are the indexes of modification points.

Operator	Translated into	Description
LineDeletion(x)	[LineReplacement(x, None)]	Delete x
LineMoving(x, y, pos=*before* or *after*)	[LineInsertion(y, x, pos), LineReplacement(x, None)]	Move x before or after y
StmtDeletion(x)	[StmtReplacement(x, None)]	Delete x
StmtMoving(x, y, pos=*before* or *after*)	[StmtInsertion(y, x, pos), StmtReplacement(x, None)]	Move x before or after y

3 GRANULARITY COMPARISON STUDY

To introduce the new features of PyGGI, the AST granularity level and the capability of accepting modification weights, we have conducted a small study of comparing search landscapes for automated program repair at different granularity levels. This Section presents the details of the experimental settings.

3.1 Research Questions

To compare the line and the AST granularity level, we ask the following research questions.

- **RQ1. Effectiveness:** which granularity level is more effective at generating patches?

To answer RQ1, we measure various metrics during the PyGGI repair runs. Success count is simply the number of successful repair runs out of the total number of trials. We also compute the ratio of valid patches, i.e., patches that do not break the syntax of the program when applied and do not go over the time limit. Finally, we count the number of unique plausible patches [19] (i.e., the patches that pass all given test cases) found for each fault.

- **RQ2. Efficiency:** which granularity level is more efficient to navigate and search?

To answer RQ2, we report the number of fitness evaluation that resulted in plausible patches, as well as the wall clock time required. For both RQ1 and RQ2, we also undertake qualitative analysis of successful and unsuccessful repair attempts to gain insights.

3.2 Subjects

We use the latest version of `sh`[2], which is a full-fledged replacement of the `subprocess` module in Python. The `sh` project is currently 3,583 LOC and comes with 156 test cases.

The study use seeded faults in `sh`: ten faulty versions of the original program have been generated by manually mutating a single statement. All seeded faults are *repairable* under the Plastic Surgery Hypothesis [2], i.e., the correct version can be obtained by rearranging the statements in the version with the seeded fault.

3.3 Fault Localisation and Test Filtering

We use Ochiai [16, 22] to provide the suspiciousness scores. Ochiai scores are computed as follows:

$$Ochiai(e_p, e_f, n_p, n_f) = \frac{e_f}{\sqrt{(e_f + n_f) * (e_f + e_p)}} \tag{1}$$

where e_p and e_f represent the number of passing and failing test cases that execute the given program element, respectively, and n_p and n_f the number of passing and failing test cases that *do not* execute the given element, respectively. The resulting Ochiai score is expected to be correlated with how likely the given element is to be faulty. Intuitively, the higher the e_f and n_p are, the more *suspicious* the element is. If the test executes the faulty element, it is more likely to fail ($\uparrow e_f$). Similarly, if the test does not execute the faulty element, it is more likely to pass ($\uparrow n_p$).

For the study, we provide pre-computed Ochiai scores as modification weights. First, lines and statements are ranked according to their Ochiai scores, with ties broken by the max tie breaker (i.e., tied elements are placed at the lowest rank). Subsequently, if there are fewer than or equal to top ten distinctive lines or statements, PyGGI considers these to form the *suspicious* set and only targets elements in this set for modifications. If, however, it is not possible to pick top ten distinctive elements, PyGGI targets all lines and statements with equal probabilities.

Since the coverage measurement tool we use reports statement coverage per physical line, Ochiai scores are computed at the line granularity level. A statement is deemed to be in the suspicious set if its first physical line is in that set. Table 3 presents the number of total modification points for line and AST granularity level, as well as the number of suspicious modification points.

The original test suite contains total 156 test cases. However, it may be that not all test cases are relevant to the fault under repair. In case where the Ochiai scores have identified top ten suspicious lines or statements, any passing test cases that do not execute these have been filtered out. If we do not have the distinct top ten target lines or statements, we do not filter out test cases. Table 4 lists the number of passing and failing test cases after test filtering. Note that the filtering method we adopted does not eliminate any test cases for fault 4 and 6.

3.4 Search Algorithm and Fitness Function

Since the test suite of `sh` contains deterministic test cases only, it is unnecessary for the search to reconsider previously evaluated patches. As such, we implement and use Tabu Search [6] by extending *localSearch* class of PyGGI. The search maintains a record of visited solutions, namely the *tabu* list, and does not go back to any in the list. The tabu search uses three modification operators: deletion, insertion, and replacement, to generate neighbouring solutions. Algorithm 1 presents the pseudo code. The `tabu` is initially an empty list (Line 1). In each iteration, the search continues to explore the neighbourhood of the current best patch, until finding a patch which is not contained in *tabu*. Once such a patch is found, the patch is added to `tabu` before the search proceeds to evaluate it (Line 19-20).

[2] https://github.com/amoffat/sh

Table 3: The number of total and suspicious modification points at each granularity level. Since AST level statements can span across multiple physical lines, the number of suspicious modifications points for AST granularity level tends to be equal or smaller than that of line granularity level.

Fault Index	Total mod. points		Susp. mod. points	
	Line	AST	Line	AST
1	3,583	1,706	4	3
2	3,583	1,707	8	5
3	3,583	1,707	9	5
4	3,583	1,707	0	0
5	3,583	1,706	9	6
6	3,583	1,706	0	0
7	3,583	1,707	4	4
8	3,584	1,708	10	8
9	3,584	1,708	10	8
10	3,585	1,708	4	3

Table 4: Number of test cases after filtering. Note that test filtering we use fails to eliminate any test case for fault 4 and 6.

Fault	Num. of test cases after filtering		
	Passing	Failing	Total
1	1	3	4
2	6	1	7
3	2	11	13
4	153	3	156
5	8	1	9
6	155	1	156
7	9	1	10
8	8	1	9
9	80	1	81
10	1	7	8

Algorithm 1: Tabu Search Algorithm

```
input  : A target program $T$
output : bestPatch, bestFitness
tabu ← [];
bestPatch ← emptyPatch($T$);
bestFitness ← fitness(bestPatch);
i ← 0;
while true do
    i ← i + 1;
    if i > 2000 or bestFitness == 0 then
        break;
    end
    while true do
        patch ← bestPatch.clone();
        if isNotEmpty(patch) and genRandomFloat([0, 1]) < 0.5 then
            patch.removeRandomEdit();
        else
            editOp = randomChoice(delete, replace, insert);
            patch.add(editOp.create($T$));
        end
        if not patch in tabu then
            tabu.append(patch);
            break;
        end
    end
    fitness ← fitness(patch);
    if fitness < bestFitness then
        bestPatch ← patch;
        bestFitness ← fitness;
    end
end
```

To evaluate each candidate solution, we use a simple fitness function for program repair. The fitness of a patch P is the number of failing test cases when we execute the given test cases on the candidate program (i.e., faulty program with the candidate patch applied). Note that we use the filtered test suite for the fitness evaluation. More formally,

$$Fitness(P) = |T_f| \tag{2}$$

where T_f is a set of failing test cases. If the application of patch breaks the syntax of the program, or the test execution exceeds the given time budget, we assign the fitness value of $+\inf$.

Because our purpose is comparing the line and the AST granularity level, we conduct the same repair experiment for both levels. We use three different modification operators: deletion, insertion, and replacement. In the case of insertion operator, we use insertion *before* only, to avoid identical patches that can be generated when using both insert before and after.

3.5 Configuration & Implementation

We execute PyGGI against the same fault for 20 times using line and AST granularity level respectively. In each attempt, the tabu search is given the budget of 2,000 fitness evaluations: the stopping criterion is either when the budget expires, or when a *plausible* patch [19] is found. Test executions time out after 20 seconds.

As described, we use Ochiai formula to compute the suspiciousness scores: the coverage data has been collected using a widely used Python coverage measurement tool, `coverage.py` [3]. The experiment has been conducted on a PC with Intel Core i7-7700 CPU and 32GB memory running Ubuntu 16.04, using Python 3.6.2. PyGGI uses `astor` [4] to manipulate the Python AST.

3.6 Results

This section presents the results from our study. Table 5 shows the comparative results from two granularity levels.

3.6.1 Effectiveness (RQ1). Among the total ten faults, we succeed to find the plausible patches for the six faults: 1, 2, 3, 7, 8, and 10. In case of the faults 4, 6, and 9, the filtered test suites still do not finish within the time-out limit of 20 seconds, resulting in test

[3]https://bitbucket.org/ned/coveragepy
[4]https://github.com/berkerpeksag/astor

Table 5: Comparison of the line and the AST granularity level in the automated repair of seeded faults in `sh`. Results are averaged from twenty repeated runs. The last column presents the number of unique correct patches in the parentheses.

Fault Index	Successful Runs		Valid Patch Rate (%)		Num. of Evaluations		Elapsed Time (s)		Plausible Patches	
	Line	AST	Line	AST	Line	AST	Line	AST	Line	AST
1	1	4	51.24	98.67	1708	976.5	720.81	588.62	1(1)	1(1)
2	20	20	30.21	84.96	602.1	11.3	1,516.91	88.07	19(0)	8(0)
3	1	1	33.31	94.94	2	745	12.68	1,818.60	1(1)	1(1)
4	0	0	0	0	-	-	-	-	-	-
5	0	0	39.88	99.88	-	-	-	-	-	-
6	0	0	0	0	-	-	-	-	-	-
7	2	9	39.20	93.29	792.5	1,041.78	2,718.60	7,107.22	2(0)	4(1)
8	20	20	55.29	99.65	14.65	14.2	15.32	18.90	3(3)	3(3)
9	0	0	0	0	-	-	-	-	-	-
10	20	19	37.5	56.06	9.6	3.37	13.87	31.10	7(6)	7(1)

failures and `+inf` fitness for all candidate patches. Consequently, PyGGI fails to generate any patches for these. For fault 5, PyGGI finds no plausible patch at all, even though the actual faulty line was chosen to be suspicious. We investigate this in more detail in Section 3.6.3.

To answer RQ1, we first compare the success rates shown in Table 5. On all repaired faults, which have at least one plausible patch, except 1 and 7, both granularity levels succeed similar number of times. However, for fault 1 and 7, PyGGI succeed roughly four times more frequently at the AST level than at the line level. We posit that this difference stems from the different nature of these faults. Let us divide the six repaired faults into two categories. The first category includes fault 1, 3, and 7: these faults require specific ingredients to be repaired. The second category includes fault 2, 8, and 10: these faults can be repaired to pass all tests without requiring specific ingredients, i.e., either by deleting it or replacing it with an ingredient that does not affect the program behaviour (such as inserting an assignment that is overwritten afterwards without being used).

In the first category, except for the fault 3, the AST level achieves a much higher success rate compared to the line level. We can relate this results with the number of total ingredients (i.e., modification points) shown in Table 3. When PyGGI pre-processes a program at the line granularity level, the number of ingredients is about twice as large as that of the AST granularity level. Consequently, the probability of finding a correct ingredient in the given budget is lower than that of the line level.

Fault 3 is the sole exception in the first category that cannot be explained by the size of the ingredient space. However, as shown in Table 4, the fault 3 has twice as many suspicious lines as fault 1 and 7. The larger the number of candidate lines to modify, the wider the search space becomes, and consequently fault 3 is much harder to fix than fault 1 and 7. As a result, it has a very low success rate of 1 out of 20 regardless of the granularity level. To summarise, the first category of faults are harder to fix due to the large search space (either due to the large number of ingredients or suspicious targets), although AST granularity level seem to be at least more successful than the line level in the case of fault 1 and 7.

On the contrary, the repair for the faults in the second category does not require specific ingredients and, as such, their success does not depend on the size of the ingredients, or even the number of total modification points. Furthermore, the nature of these faults means that it is easier to repair them within the given budget, compared to the faults in the first category.

Since the two levels generate a plausible patch for the same fault sets, we tried to establish the correctness of the found patches. First, we executed the un-filtered test suite against all patches: all of the generated plausible patches passed the entire test suites. Next, we manually investigated the correctness of the plausible patches to provide qualitative answers to RQ1. We define a *correct* patch as a patch that modifies the version with the seeded faults to be semantically equivalent to the original one again. The last column of Table 5 presents the number of unique correct patches in the parentheses.

```
1 @@ -1583,7 +1583,7 @@
2          handler_to_inspect = handler.func
3 
4      if inspect.ismethod(handler_to_inspect):
5 -        implied_arg = 0
6 +        implied_arg = 1
7          num_args = get_num_args(handler_to_inspect)
8 
9      else:
```

Figure 4: Correct patch generated for the fault 7: Replacing the 736th statement with the 740th statement.

Only the AST granularity level could generate a correct patch for the fault 7, which is shown in Figure 4. It repairs the faulty program to be the same as the original program. Because the required ingredient (Line 5) originally has a different indentation level from the target modification point (Line 4), it can be repaired only at the AST granularity level. The attempt to perform the same insertion at the line granularity level results in a syntax error, as the insertion would copy the indentation whitespace character along with the contents of the line, which results in a violation of the indentation rule Python uses to organise its code. Note that this would not have caused any problem in languages such as Java or C.

Manual inspection of patches generated for fault 2, 8, and 10 provides interesting insights into test-based program repair. Fault 2

```
@@ -741,7 +741,7 @@
             pipe = OProc.STDERR

         if call_args["iter_noblock"] == "out" or
     call_args["iter_noblock"] is True:
-            pipe = OProc.STDOUT
+            pipe = OProc.STDERR
         elif call_args["iter_noblock"] == "err":
             pipe = OProc.STDERR
```

Figure 5: Fault 2.

redirects some of the program output to the STDERR as in Figure 5. There is no test case to check that the program output should be made to STDOUT. Consequently, PyGGI can replace `pipe = OProc .STDERR` with any side effect free statement to generate plausible patches, but fails to generate a correct patch.

Fault 8 and 10 illustrates an interesting contrast between the line and AST granularity level. Figure 6 and 7 present these faults.

```
@@ -1525,6 +1525,7 @@
             elif v is False:
                 pass
             elif sep is None or sep == " ":
+                processed.append(encode(k))
                 processed.append(encode(prefix + k))
                 processed.append(encode(v))
             else:
```

Figure 6: Fault 8.

```
@@ -2574,6 +2574,8 @@
             raise NotYetReadyToRead
         if chunk is None:
             raise DoneReadingForever
+        else:
+            raise DoneReadingForever
         return chunk
     return fn
```

Figure 7: Fault 10.

Fault 10 in Figure 7 is generated by inserting two lines of code (and consequently forces the program to always raise an exception), whereas fault 8 inserts only one line of code. In both cases, inserted lines need to be removed for repair. However, fault 10 can be repaired by either removing `else:` or replacing `raise DoneReadingForever` with something without any side effects. The possibility of using the replace-with-no-side-effect tactic, combined with two possible modification points, provides a much productive search space for fault 10, because PyGGI can simply replace either line with any of the single line comments.

To summarise, the AST granularity level modifications are more reliable than the line granularity level in terms of finding both plausible and correct patches. Furthermore, the statement-level manipulation can generate a correct patch that cannot be produced with the line-level manipulation.

3.6.2 *Efficiency (RQ2).*

To compare the efficiency, we report the number of fitness evaluations and the wall clock time required until finding the first plausible patch in each trial. The average values are shown in Table 5.

For all the faults except 3 and 7, the AST level requires fewer fitness evaluations to find a plausible patch compared to the line level. On the contrary, the line level spends less time than the AST level except for fault 1 and 2. Since the AST granularity level yields a significantly larger number of valid (i.e., syntactically intact) patches, more AST level patches (including unsuccessful ones) are evaluated by test execution, resulting in longer overall execution time of PyGGI. In comparison, the majority of patches created by the line level modifications are syntactically incorrect, and can be evaluated without invoking test executions.

In summary, the fitness evaluation required to find a plausible patch is generally less at the AST level, but the overall execution time is longer due to the larger number of test executions.

3.6.3 *Case study of fault 5.*

In the experiment, PyGGI could not find any plausible patch for fault 5. To investigate the cause, we look at how the fault was generated. Figure 8 shows the difference between the original and the version with fault 5 seeded.

```
1  @@ -1526,7 +1526,7 @@
2                  pass
3              elif sep is None or sep == " ":
4                  processed.append(encode(prefix + k))
5  -               processed.append(encode(v))
6  +
7              else:
8                  arg = encode("%s%s%s%s" % (prefix, k,
       sep, v))
9                  processed.append(arg)
```

Figure 8: Fault 5.

The statement at Line 4 was included in the set of suspicious points. However, this fault is impossible to be repaired because of the nature of *insert* operator that we used. The insert operator employed in this study inserts the ingredient `before` the modification points, but, as shown in the figure, the deleted statement (Line 5) must be inserted `after` the point.

Therefore, we conducted an additional experiment for fault 5 using *insert after* instead of *insert before* for a fair comparison between the line and the AST granularity level. At the AST level, a plausible patch was generated in five out of twenty runs, and they contained a correct patch. However, no plausible patch was found at the line granularity level runs.

4 RELATED WORK

Genetic Improvement (GI) has received much attention [17] since GenProg demonstrated that Genetic Programming can be harnessed to automatically patch program faults using only dynamic analysis (i.e., test executions) as the fitness function [5, 20]. Automated Program Repair (APR) is now a rapidly advancing area [10, 12–15]. Langdon and Harman showed that the same GP based approach can be used to improve non-functional properties of software [11]; Petke et al. showed how software systems can be specialised for certain classes of inputs [18]. A large portion of existing literature can be categorised as Generate and Validate method, i.e., generating many candidate solutions using either evolutionary or other metaheuristic algorithms, and validate whether they are acceptable, usually by test executions.

PyGGI aims to provide a general framework on which GI techniques can be developed at the API level. There are open source implementations of GI techniques. For example, ASTOR [13] is a suite of both Java re-implementations of existing APR approaches and new ones developed by the authors. While ASTOR does have some common infrastructure shared by different GI implementations, the main design purpose is not to expose the infrastructure to the end user. GIN [21] is a Java framework whose design aim is the closest to ours, i.e., to be used by the end user. The main differences are 1) the choice of implementation language (we choose Python as we think it is more appropriate for fast prototyping than Java) and 2) the suggested use case (we aim to provide API-level usage, whereas GIN requires the end user to more directly interact with its own source code). Notably, Haraldson et al. deployed a Python-based GI implementation that is integrated into a live production system called Janus Manager [9]. While PyGGI and the work of Haraldson et al. share the same target language, Python, PyGGI aims to provide a general framework for GI whereas Janus Manager is a GI process integrated to a specific application.

5 CONCLUSION

We present a new version of PyGGI, a Python general framework for Genetic Improvement. It has been designed to be used at the API level: the new version supports AST granularity level modifications for Python, i.e., modifications that handle statement nodes in the AST rather than statements marked by physical lines, which was what the previous version depended on. We use the new features to conduct a study of comparing search landscapes at different granularity levels for Automated Program Repair, using a real world Python open source project and seeded faults. The results show that the AST granularity level and the corresponding search landscape can be more effective (i.e., more frequently results in plausible patches) because they result in smaller ingredient spaces (i.e., fewer modification points). However, because the AST granularity level tends to result in much larger number of syntactically intact candidate patches, it may also lead to less efficiency and longer execution time due to the higher number of test executions.

ACKNOWLEDGEMENT

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (No. 2017M3C4A7068179).

REFERENCES

[1] Gabin An, Jinhan Kim, Seongmin Lee, and Shin Yoo. 2017. PyGGI: Python General framework for Genetic Improvement. In *Proceedings of Korea Software Congress (KSC 2017)*.

[2] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 306–317.

[3] David Binkley, Daniel Heinz, Dawn Lawrie, and Justin Overfelt. 2014. Understanding LDA in source code analysis. In *Proceedings of the 22Nd International Conference on Program Comprehension*. ACM, 26–36.

[4] François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: A Python Framework for Evolutionary Algorithms. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '12)*. ACM, New York, NY, USA, 85–92.

[5] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A Genetic Programming Approach to Automated Software Repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09)*. ACM, New York, NY, USA, 947–954.

[6] Fred Glover and Manuel Laguna. 2013. Tabu Search. In *Handbook of combinatorial optimization*. Springer, 3261–3362.

[7] Nicolas Gold, David Binkley, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2017. Generalized Observational Slicing for Tree-Represented Modelling Languages. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2017)*. 547–558.

[8] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering*. 3–13.

[9] 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 *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. ACM, New York, NY, USA, 1513–1520.

[10] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 802–811.

[11] W.B. Langdon and M. Harman. 2015. Optimizing Existing Software With Genetic Programming. *Transactions on Evolutionary Computation* 19, 1 (2015), 118–135.

[12] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 727–739.

[13] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*. https://doi.org/10.1145/2931037.2948705

[14] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 448–458.

[15] S. Mechtaev, J. Yi, and A. Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701.

[16] A Ochiai. 1957. Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. *Bulletin of the Japanese Society of Scientific Fisheries* 22, 9 (1957), 526–530.

[17] J. Petke, S. Haraldsson, M. Harman, w. langdon, D. White, and J. Woodward. 2017. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* PP, 99 (2017), 1–1. https://doi.org/10.1109/TEVC.2017.2693219

[18] 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 *17th European Conference on Genetic Programming (LNCS)*, Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo Garcia-Sanchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim (Eds.), Vol. 8599. Springer, 137–149.

[19] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 24–36.

[20] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE '09)*. IEEE, Vancouver, Canada, 364–374.

[21] David R White. 2017. GI in no time. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, 1549–1550.

[22] W. E. Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (August 2016), 707.