| 저자 (Authors) | Gabin An, Jinhan Kim, Seongmin Lee, Shin Yoo |
| Source | 한국정보과학회 학술발표논문집, 2017.12, 536-538(3 pages) |
| Publisher | The Korean Institute of Information Scientists and Engineers |
| URL | http://www.dbpia.co.kr/journal/articleDetail?nodeId=NODE07322214 |
| Accessed | KAIST 143.***.244.75 2019/06/18 14:22 (KST) |

저작권 안내
DBpia에서 제공되는 모든 저작물의 저작권은 원저작자에게 있으며, 누리미디어는 각 저작물의 내용을 보증하거나 책임을 지지 않습니다. 그리고 DBpia에서 제공되는 저작물은 DBpia와 구독계약을 체결한 기관소속 이용자 혹은 해당 저작물의 개별 구매자가 비영리적으로만 이용할 수 있습니다. 그러므로 이에 위반하여 DBpia에서 제공되는 저작물을 복제, 전송 등의 방법으로 무단 이용하는 경우 관련 법령에 따라 민, 형사상의 책임을 질 수 있습니다.

Copyright Information
Copyright of all literary works provided by DBpia belongs to the copyright holder(s) and Nurimedia does not guarantee contents of the literary work or assume responsibility for the same. In addition, the literary works provided by DBpia may only be used by the users affiliated to the institutions which executed a subscription agreement with DBpia or the individual purchasers of the literary work(s) for non-commercial purposes. Therefore, any person who illegally uses the literary works provided by DBpia by means of reproduction or transmission shall assume civil and criminal responsibility according to applicable laws and regulations.
PYGGI: Python General Framework for Genetic Improvement

Gabin An  
agb94@kaist.ac.kr
Jinhan Kim  
jinhankim@kaist.ac.kr
Seongmin Lee  
bohrok@kaist.ac.kr
Shin Yoo  
shin.yoo@kaist.ac.kr
KAIST, Republic of Korea

Abstract

We present Python General Framework for Genetic Improvement (PYGGI, pronounced ‘pgg’), a lightweight general framework for Genetic Improvement (GI). It is designed to be a simple and easy to configure GI tool for multiple programming languages such as Java, C, or Python. Through two case studies, we show that PYGGI can modify source code of a given program either to improve non-functional properties or to automatically repair functional faults.

1. INTRODUCTION

Genetic Improvement (GI) is an emerging research field that aims to apply meta-heuristic optimisation techniques (such as Genetic Programming) to software systems in order to automatically improve its functional and non-functional properties [1].

It is still a young and emerging field and, as such, lacks framework support for researchers and practitioners alike. Existing work mostly relies upon provisional and, more importantly, language-specific implementations. For example, the recently introduced Gin [2] is implemented for a specific target language and instruments only a single source file at once. Consequently, one has to directly modify the source code of Gin itself to apply it to projects that consist of multiple files, not to mention projects that are implemented in more than one programming language.

In this paper, we introduce Python General framework for Genetic Improvement (PYGGI), which is designed as a configurable and easy to use GI framework. The initial release includes line-based code modification operators (similar to those used by the widely studied automatic program repair tool, GenProg [3]) and a simple local search heuristic. We plan to extend PYGGI so that it supports more sophisticated search heuristics and modification operators. Using the initial version of PYGGI, we present two proof-of-concept case studies: genetic improvement of non-functional properties, and automated program repair. Considering that Python is widely regarded as a high productivity language that is ideal for fast prototyping, we anticipate that PYGGI will help researchers and practitioners to quickly implement and evaluate their future GI ideas, while writing as few lines of code as possible.

2. OVERALL ARCHITECTURE

PYGGI manipulates target source code files that are specified by a configuration file, and observes the impact of manipulation by running a test script written by the user. The test script should print output in the predefined, PYGGI-recognisable format. Modifications of the original program source code is represented as a patch. For current implementation, PYGGI only uses code lines found in the program under improvement as the ingredients of patches it generates. This is achieved by our choice of modification operators, which are deletion, copying, and replacement. Our operator set is similar to the widely studied automated bug patch tool, GenProg [2].

2.1 PYGGI CLASSES

![Class Diagram of PYGGI](image)

Program encapsulates the original source code. Currently, PYGGI stores the source code as a list of code lines, as lines are the only supported unit of modifications. For modifications at other granularity levels, this class needs to process and store the source code accordingly (for example, by parsing and storing the AST).

Patch is a sequence of edits: deletion, copying, and replacement. During search iteration, PYGGI modifies the source code of the target program by applying a candidate patch. Subsequently, it runs the test script to collect dynamic information, such as the execution time or any other user-provided properties, via the predefined format that PYGGI recognises. Finally, the fitness value of the patch is calculated using the user-provided fitness function, written in Python.
An Edit class defines an atomic edit operator, such as deletion, copying, and replacement. It is possible for the user to define new edit operators by writing custom classes.

TestResult stores the result of the test suite executions on the original or patched source code. It records whether compilation succeeded, the elapsed execution time, as well as any other user-defined test results.

2.2 LOCAL SEARCH ALGORITHM

Algorithm 1 Pseudocode for Local Search

<table>
<thead>
<tr>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program, $P$, Random Neighbour Selection, $N$, Fitness Function, $F$, Patch Validator, $V$, Termination Criterion, $E$,</td>
</tr>
</tbody>
</table>

```java
1: bestPatch ← [] // empty patch
2: bestFitness ← $F(apply(P, bestPatch))$
3: i ← 0
4: while true do
5: i ← i + 1
6: if $E(i, bestPatch)$ then
7: break
8: end if
9: patch ← $N(bestPatch)$
10: if ¬$V(patch)$ then
11: continue
12: end if
13: fitness ← $F(apply(P, patch))$
14: if ¬isBetter(fitness, bestFitness) then
15: continue
16: end if
17: bestPatch ← patch
18: bestFitness ← fitness
19: end while
20: return bestPatch
```

Figure 2 Local search algorithm

Figure 2 presents the local search heuristic currently provided by PYGGI. It starts from a randomly generated candidate solution and iteratively moves to its neighboring solution with a better fitness value by making small local changes to the candidate solution.

3. CASE STUDY

This section describes two case studies that use a small benchmark program, triangle, which has been widely studied in automated test data generation [4] as well as in GI [1, 5]. It takes as input three integers, each of which represents a length of the sides of a triangle: it outputs the type of triangle that can be formed. Table 1 shows the test suite for the program.

As we described in Section 2.2, the base GI algorithm in both case studies is local search. The local change was either adding a random edit to the tail of an edit sequence or removing one random edit from an edit sequence. The random edit is chosen from the three operators: line deletion, copying, and replacement. The experiments were conducted on a PC with Intel Core i7-7700 CPU and 32GB memory running Ubuntu 16.04.

### Table 1 The original test suite for Triangle

<table>
<thead>
<tr>
<th>Type</th>
<th>Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid</td>
<td>(1, 2, 9), (-1, 1, 1), (1, -1, 1), (1, 1, -1), (100, 80, 1000)</td>
</tr>
<tr>
<td>Equilateral</td>
<td>(1, 1, 1), (100, 100, 100), (99, 99, 99)</td>
</tr>
<tr>
<td>Isosceles</td>
<td>(100, 90, 90), (1000, 900, 900), (3, 2, 2), (30, 16, 16), + (2, 2, 1)</td>
</tr>
<tr>
<td>Scalene</td>
<td>(5, 4, 3), (1000, 900, 101), (3, 20, 21), (999, 501, 600)</td>
</tr>
</tbody>
</table>

3.1 IMPROVING NON-FUNCTIONAL PROPERTIES

First, we present a proof of concept that PYGGI can use a non-functional property as a fitness function for its local search. For this, we inject a statement that intentionally delays execution into the benchmark program and check whether PYGGI can correctly remove this when execution time is given as the property to improve. PYGGI successfully deletes the line `delay();`, which calls another method that makes the current thread sleep for 50 milliseconds. The search finishes either after evaluating 1,000 candidate patches or when the fitness value falls below a threshold, in this case, 100 milliseconds. Figure 3 shows the patch generated by PYGGI.

The experiment was repeated 50 times: PYGGI successfully generated correct patches within 1,000 fitness evaluations for all 50 runs. Table 2 reports the average number of Patches Searched (NPS) (i.e. number of candidate patches considered until the solution was found), and time elapsed until the solution is found.

```
7: public static TriangleType classifyTriangle(int a, int b, int c){
8:     return;
9:     delay();
10:    // Patch 1 */
11:    delay();
12:   }
13:   if (a > b) { // Sort the sides so that a <= b <= c
14:      int tmp = a;
15:   a = b;
```

Table 3 sample/Triangle_delay/Triangle.java

<table>
<thead>
<tr>
<th>Average NPS</th>
<th>Time Elapsed(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>102.7</td>
<td>70.8</td>
</tr>
<tr>
<td>98.0</td>
<td>59.5</td>
</tr>
</tbody>
</table>

3.2 AUTOMATED PROGRAM REPAIR

The second case study investigates automated program repair. We inject a fault by replacing `int tmp = a;` with `int tmp = b;` the resulting program does not pass all test cases. Since the original line is available from other parts of the triangle program, this fault can be repaired using the current implementation of PYGGI. Unlike the
execution time, the functional behaviour of the triangle program is
deterministic, and test results remain the same for the same patch.
Consequently, we modified the local search to maintain a tabu list,
preventing the search from revisiting a patch that has already been
evaluated before. The fitness function is the number of passing test
cases; the search terminates when a plausible patch [6], i.e., a patch that
makes all test cases pass, is found or after one hour is passed.

```c
13: if (a > c) {
14:     int tmp = b; // original: int tmp = a;
15: /* Patch 1: Correct patch */
16:     - int tmp = b;
17:     + int tmp = a;
18: /* Patch 2: Plausible but incorrect patch */
19:     + } else if (a == b && b == c) {
20:     - a = c;
21:     + c = tmp;
22: }
23: }
```

Figure 4 Generated Patches for triangle Fault

Table 2 Experimental results for Section 3.2 (50 repetitions)

<table>
<thead>
<tr>
<th></th>
<th>NPS</th>
<th>Time Elapsed(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>915.8</td>
<td>454.9</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>585.5</td>
<td>87.6</td>
</tr>
</tbody>
</table>

PYGGI generates two plausible patches during initial repair, which
are listed in Figure 4. Note that the patch 2 is plausible but semantically
incorrect, whereas the patch 1 is correct. We manually generated a new
test input, (2, 2, 1), that can differentiate the original and the buggy
program. When the new input is added to the test suite, PYGGI avoids
overfitting to the plausible patch. We repeat the program repair run for
50 times using the updated test suite, the results of which is shown in
Table 3. PYGGI succeeds for all 50 runs, after evaluating about 900
candidate patches in 7.5 minutes on average.

4. DISCUSSIONS

The initial implementation of PYGGI uses physical lines as the
unit of patches. However, we note that certain faults may require finer-
gained modifications, and plan to investigate AST-level edit operators.

Existing Generate and Validate techniques are susceptible to
plausible repairs, i.e. overfitting to insufficient test suites [7]. PYGGI is
not exempt from this inherent limitation, as in the case of the plausible
patch we described in Section 3.2. Future work will consider the use of
automated test data generation to reduce the risk of insufficient test
cases and overfitting.

Finally, the initial implementation of PYGGI does not use any fault
localisation technique for functional improvements. Use of fault
localisation can reduce the size of fault space, which will increase both
the effectiveness and efficiency of PYGGI’s functional improvement.

5. CONCLUSION & FUTURE WORK

We introduce PYGGI, a Python General framework for Genetic
Improvement. Its initial version is capable of improving a non-
functional property or repairing a bug inside a program, using a local
search algorithm. We conduct and report results of two proof-of-
concept case studies. Future work will consider AST-level code
manipulations, fault localisation, and orchestration of automated test
data generation and automated program repair.

6. ACKNOWLEDGEMENT

This research was supported by the MSIT (Ministry of Science and
ICT), Korea, under the National Program for Excellence in SW
supervised by the IITP (Institute for Information & communications
Technology Promotion) (2016-0-00018). 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).

7. REFERENCES

Woodward. Genetic improvement of software: a comprehensive survey.
finding patches using genetic programming. In Proceedings of the 31st
IEEE International Conference on Software Engineering (ICSE ’09),
algorithms. A plug and play research tool. Technical Report CS-07-14,
Department of Computer Science, University of Sheffield, 2007.
[5] D. White, A. Arcuri, and J. Clark. Evolutionary improvement of
–538, August 2011.
plausibility and correctness for generate- and-validate patch generation
systems. In Proceedings of the 2015 International Symposium on
Software Testing and Analysis, ISSTA 2015, pages 24–36, New York,
NY, USA, 2015. ACM.
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, pages 532–543, New York,
NY, USA, 2015. ACM.