

CS454 FALL 2019

AI BASED SOFTWARE ENGINEERING

---

# Search-based Test Case Generation for Code Plagiarism Checkers

---

*Author(Github ID):*

Seungjun CHUNG(sjoon2455)

Doheon HWANG(hdh112)

Adrian STEFFAN(adriansteffan)


Juyeon YOON(greenmonn)

Simon ZOCHOLL(SimonZocholl)

*Supervisor:*

Prof. Shin Yoo

December 20, 2019

 <https://github.com/cs494-team4/search-based-plagiarism>

## Abstract

We propose a search-based test case generation framework, **PlagGen** for program-similarity measurement tools utilizing automated refactorings. We evaluated our approach against two similarity detection tools, **Moss**[1] and **pycode-similar**[2]. A custom refactoring engine was implemented covering the following levels and scopes: expression-level, function-level, class-level, and control-flow manipulation. NSGA-II was utilized for the genetic algorithm with custom representation, operators, and fitness. The generated test cases achieved considerably low similarity scores on similarity detection tools, but plagiarism could be easily detected when the test cases were proofread by humans. Certain refactoring types were especially effective such as adding guard clauses or refactoring conditional branches.

## 1 Introduction

Software plagiarism is a serious copyright violation that infringes others' intellectual properties. To combat this issue, various plagiarism detection tools based on detecting the similarity of source code files are deployed. These tools are commonly used in academic teaching to automatically deal with large quantities of code bases, as it would be exhaustive for a limited human faculty to review. Yet, there is limited research to measure the accuracy of the inferences and the extent of plagiarism that these tools detect.

Previous research[3] have compared code similarity tools that detect on programs written in Java with randomly obfuscated benchmark dataset. In our research, we specifically generate test cases from programs written in Python, and the generated test cases are guided by GA to find more critical scenarios for the target plagiarism detector. We approach this as a multi-objective problem as to maximally decrease the similarity with the minimum number of refactorings, in order to prevent code explosion, mirror real world settings more closely. By producing quality test cases rather than randomly generated ones, we can investigate which refactoring affects similarity detection the most. We can then look at these refactorings to identify the weaknesses and propose possible improvements of the detection tools.

## 2 System Design

**PlagGen** consists of two main components: First, the GA that builds a Pareto front by coming up with reasonably short sequences of refactorings that achieve a low similarity score when compared to the base. Second, a fitness function to evaluate these sequences. To do this, the fitness function needs to apply a sequence of refactorings to the initial codebase and then access plagiarism detection tools to gauge the similarity score of the applied sequence. The following paragraphs will give an overview of the structure and control flow of the system enabling this as well as the used refactoring engine and similarity assessment, while the details of the GA will be looked at separately.

### 2.1 Overview

The structure adheres to object-oriented principles. The general structure of our program is visualized in figure 1.

Note that all UML diagrams shown here are simplified with the intent of providing a general overview of the system while abstracting details like exact call parameters or object factories. As is seen, the fitness function has two main abstract subsystems: (1) the **Refactorer**, responsible for applying sequences of refactorings to a given codebase using a variety of operators, and (2) the **SimilarityClient**, responsible for accessing the judgment of a software similarity assessment tool.

We choose an “un-Pythonic” approach to the project structure, explicitly using abstract classes (leveraging Python’s abstract base class) and factories to switch between implementations of both the **SimilarityClient** and the **Refactorer**. This has two main reasons. As we faced issues with multiple different external services early on (**Code-Imp** not being available, **MOSS** having server issues and proving to be a bottleneck) we built the software with modularity in mind. This way, switching out one part of the system and implementing a new one was easy, which came in hand when we had to include **pycode-similar** as a second similarity engine. As we want to evaluate different plagiarism tools for future work, being able to easily include new ones is also an important consideration. This modularity was also motivated by the team size of five members during development. As parts of the system were independent and functionality could easily be dummied for testing, this structure enabled simultaneous work and reduced dependencies between the

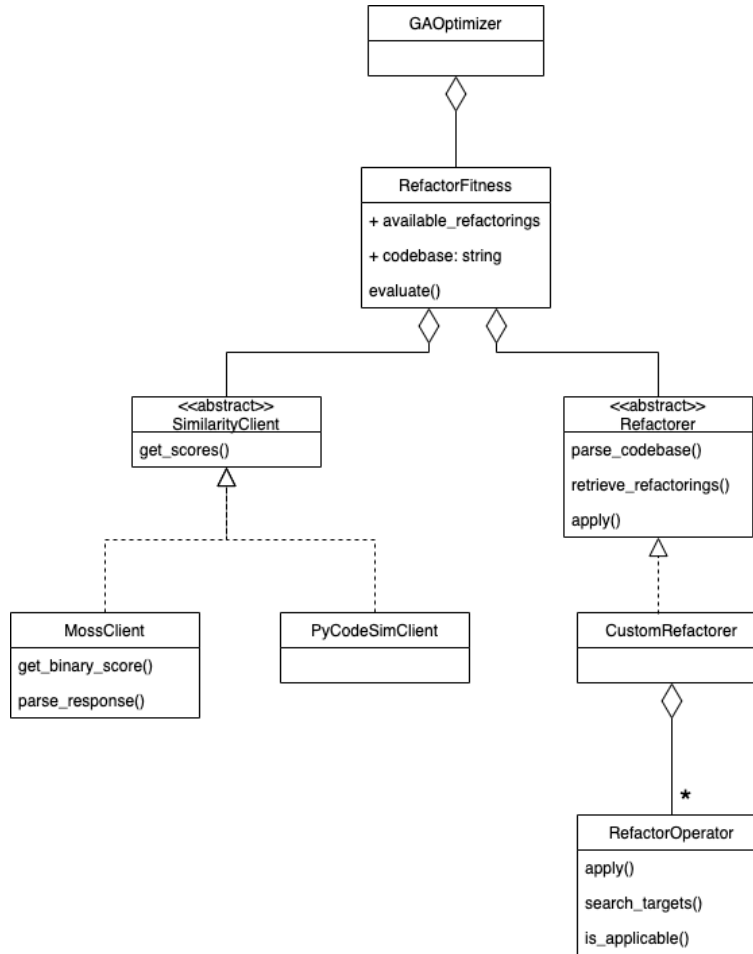


Figure 1: PlagGen Class Diagram

work of different team members.

The structure also makes extending the system for future work or via independent contributions easier. Looking at the docstring of the abstract classes gives the information needed about the kind of data these new implementations need to expose. The newly created classes just need to be specified in the corresponding factory after implementation. Currently, we are able to integrate new implementations for both the **SimilarityClient** and the **Refactorer** this way, with the **Operators** for our **CustomRefactorer** being extendable in a similar way (and of course, replacing the metaheuristic

is done by accessing the fitness function from a different context).

## 2.2 System Flow

The fitness system has to expose two functionalities to the genetic algorithm. First, it has to inform the GA what different kinds of refactorings are available and what the possible targets are in a given codebase. During the runtime of the GA it also has to evaluate the fitness value of given sequences.

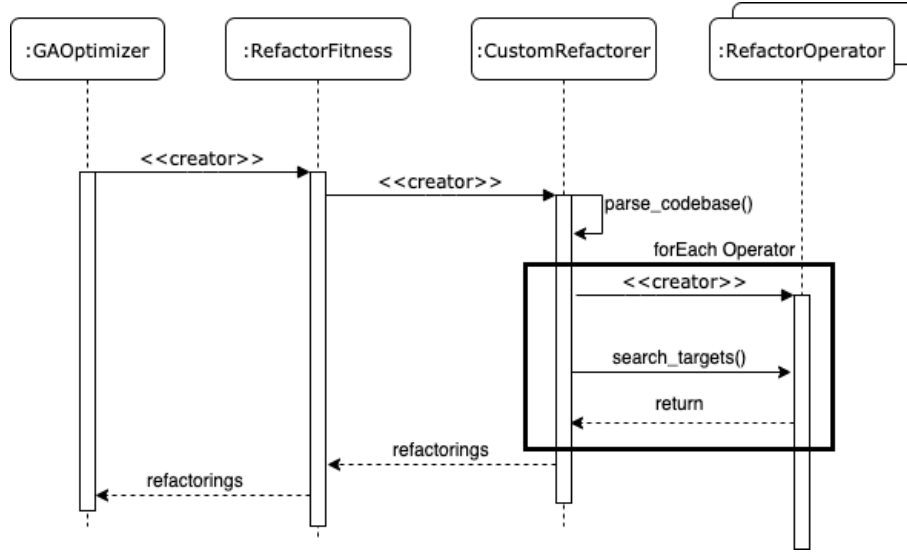


Figure 2: Sequence Diagram on Identifying Possible Refactorings

Figure 2 shows how the GA gets access to the list of possible refactorings. After the fitness function constructs the `CustomRefactorer`, the refactorer instantiates objects of all possible refactoring operators that are implemented. For each of these, the syntax tree is walked and all possible targets for the given refactoring are identified. The `CustomRefactorer` then aggregates this information in a dictionary and passes the possible refactorings and targets back to the fitness function, which then exposes the information to the GA.

Figure 3 shows the program flow for a singular fitness evaluation. After `evaluate()` gets called on the `RefactorFitness` object of the GA, it instructs the `CustomRefactorer` to generate the new code sample by calling the `apply()` method. The refactorer then takes the codebase and applies the

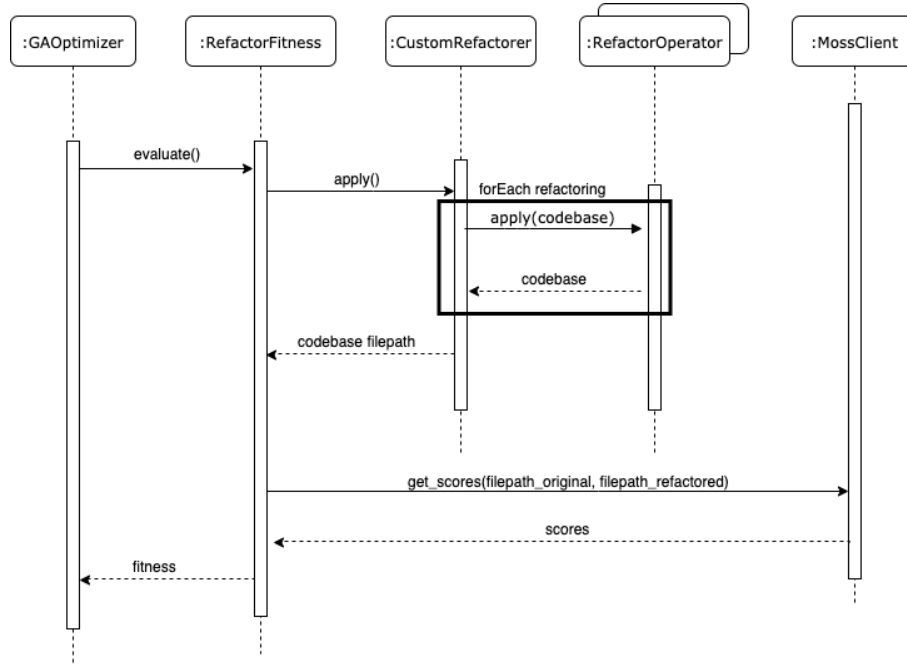


Figure 3: Sequence Diagram on Fitness Evaluation

given refactorings in sequential order (ignoring those that are no longer valid due to earlier refactorings). The newly generated codebase gets saved and a file path is passed back to the fitness function, which then sends this path together with one pointing to the original codebase to the **SimilarityClient**. The **SimilarityClient** calculates the plagiarism/similarity score of the two samples, either by sending a request to **MOSS** or utilizing **pycode-similar**. The similarity value is then passed back to the GA, which can use this information to guide the search for better refactoring sequences. Note: The GA actually passes a list of refactoring sequences instead of a singular one for evaluation to ensure compatibility with future parallelization. This detail was omitted from the diagrams to make the figures easier to comprehend.

## 2.3 Components

### 2.3.1 Similarity Client

As this project wants to explore if search could be used to find possible weaknesses in software plagiarism detection tools, choosing an appropriate

platform to test our approach against is important. We decided to use Stanford’s Measurement of Software Similarity[1] as it is one of the most widely used plagiarism detection tools in academia and is also freely available after a simple signup process. **MOSS** tokenizes the code instead of looking at the raw strings and is thereby hard to trick by renamings or simple reorderings. **MOSS** is server-based: after sending in a request with multiple source code files and your user key, **MOSS** answers with an HTTP response. By parsing this HTTP response (handled in `MossClient.py`) we can extract the similarity score that **MOSS** outputs.

During the development of our genetic algorithm, we faced some issues regarding **MOSS**. First, as we had to access **MOSS** via a web server, the response time served as a severe bottleneck during GA runs, as a singular fitness evaluation took several seconds. While we designed our system architecture with possible parallelization of these requests in mind (utilizing multiple user ids), time limitations made us prioritize other features over this. On top of that, this would have not fixed the frequent server issues that **MOSS** experienced, preventing us from testing the GA for several hours at a time. So while we still consider **MOSS** to be one of the main targets to evaluate our algorithm against, we had to come up with a different solution to make the development of the search algorithm efficient. To assess software similarity, we looked into `pycode-similar`[2]. This library extracts the syntax tree of the given code samples and calculates a similarity score by simply applying line diff algorithm to formatted AST string lines. As we could run this evaluation offline, it enabled us to reiterate on our GA more often.

- **MOSS**
  - ◊ Accessed via a web server
  - ◊ Language-independent
  - ◊ Observes tokenized code string
- `pycode-similar`
  - ◊ Can be computed in local environment
  - ◊ Python-specific
  - ◊ Uses parsed AST to measure similarity

### 2.3.2 Refactoring Operator

- Expression-level

- ◇ Change for-loop to while-loop (ForToWhile)
- ◇ FormatToStringConcat (FormatToStringConcat)
- ◇ Change pow to \*\* (PowToOperator)
- ◇ Change \*\* to pow (OperatorToPow)

The for-loop to while-loop refactoring, replaces a for loop with a while loop that iterates over the iterable object, by increasing a counter variable `i`. An example for this refactoring is before:

```
for x in [1, 2, 3]:  
    print(1)
```

after:

```
i = 0  
while i < len([1, 2, 3]):  
    x = [1, 2, 3][i]  
    print(1)  
    i += 1
```

The FormatToStringConcat refactoring is applicable to every string every format string. It refactors the format string by inserting the variables casted to strings directly in to a concatenation of strings. An example for this refactoring is before:

```
text = "{}_foo_{}".format(1,5,"bar")
```

after:

```
text = str(1) + "_foo_" + str(5) + "bar"
```

The pow to \*\* (PowToOperator) replaces a pow function call with a \*\* statement. And the \*\* to pow operator does the same thing, the other way around. An example for those refactorings is before:



```
a = pow(2,3) # pow to **  
b = 4**7 # ** to pow
```

after:

```
a = 2**3 # pow to **  
b = pow(4,7) # ** to pow
```

## • Function-level

- ◇ Fill in default keywords
- ◇ Fill in default arguments
- ◇ Arithmetic expression to n-ary function
- ◇ Comparison to n-ary function
- ◇ Change order of parameters

The Fill in default arguments and Fill in default keywords Refactoring are applicable to functions that are defined in the same syntax tree. They searches for key word parameters that have a default value, that is not overwritten by the function call parameters. Those default parameters are then made explicit in the function call. The keyword parameters are always appended to the parameter list, to ensure that the refactored function call is still valid python. An example for this refactoring is as follows.

before:

```
def foo(a=1, b=2, c=3):  
    return a + b + c  
  
foo(3, b=4)
```

after:

```
def foo(a=1, b=2, c=3):  
    return a + b + c  
  
foo(3, b=4, c=3)
```

The refactoring Arithmetic expression to n-ary function and Comparison to n-ary function change a Arithmetic expression or a Comparison by defining and calling a function and returns the value of the expression. The function is inserted below the `import` statements and called with the appropriate parameters. So that instead of evaluating the expression directly, the function gets evaluated and the return value is taken as value of the expression. An example for Arithmetic expression to n-ary function refactoring is following, the Comparison to n-ary works similar.

before:

```
a = 1 + 2 + b + c
```

after:

```
def wrjqxcampo(a, b, c, d):  
    return a + b + c + d  
  
a = wrjqxcampo(a, b, c, d)
```

The refactoring Change order of parameters reverses is applicable to function calls where the function is defined in the same syntax tree. It reverses the order of the parameters in the function definition, and appropriately also the order of the parameters in the function calls. An example for this refactoring is as follows.

before:

```
def foo(a, b):  
    return a + b  
  
foo(1, 2)
```

after:

```
def foo(b, a):  
    return a - b  
  
foo(2, 1)
```

- **Class-level**

- ◇ Change `static` method to `instance` method (`StaticToInstance`)
- ◇ Push down method from super class to subclass (`MethodPushDown`)

The **Static to Instance** refactoring switches a static method of a class into an instance method. Then the method gets more bound to the class, and the similarity score slightly drops with this change. Yet, the refactoring only changes the definition of the static method to instance. An example for this refactoring is as follows.

before:

```
class Foo(object):  
    @staticmethod  
    def foo():
```

after:

```
class Foo(object):  
    def foo(self):
```

The **Method Push Down** refactoring selects one of the user-defined methods in a super class, and copies the method into its subclasses only when the subclasses do not have the method defined by themselves yet. Right now, this refactoring only searches super and subclasses that are both defined in the same single codebase.

This refactoring drops the similarity score every time a method from a super class is copied into the subclass. For instance, for the instance above, the similarity score drops from 99 to 95. However, it was difficult to find a single file codebase with super and subclasses both defined in it. Eventually, the target codebases we used in the experiment did not have both super and subclasses within the codebase, so the refactoring was not applied.

If this feature is extended to recognize super/subclasses in other codebases, and if the system is able to input multiple codebases, this refactoring is estimated to be powerful in lowering the similarity score. While this refactoring lowers the similarity, it is unfavorable that same code(target method) gets copied excessively.

- **Control Flow Manipulation**

- ◇ Split and conditional (`SplitAndConditional`)

- ◇ Split or conditional (`SplitOrConditional`)
- ◇ Merge nested if statement (`MergeNestedIfStatement`)
- ◇ Add else after return, break, continue (`AddElseAfterReturnBreakContinue`)

We had an intuition that each simple coding-test-style *mistake* could be described as a mutation of the code. For example, we could define the mistake of writing `>` instead of `>=`, or vice versa as a mutation operator.

The Split and conditional operator is applicable to any if conditional that is structured like `if a and b:`. It splits the expression in two parts, and nests `b` in an `If a:` statement. The body's and `else` statements get readjusted appropriately. An example for those refactoring is as follows.

before:

```
if a and b:
    print(1)
else:
    print(2)
```

after:

```
if a:
    if b:
        print(1)
if not (a and b):
    print(2)
```

The Split or conditional operator is applicable to any if conditional that is structured like `a or b:`. It changes the if statement by duplicating the body and inserting it after `if a:` and after `elif b:`. At the time of writing this report, we discovered that it currently discards the else part of the if statement. An example for those refactoring is as follows.

before:

```
if a or b:
    print(1)
```

after:

```
if a:
    print(1)
```

```
elif b:
    print(1)
```

The Refactoring Merge nested if statement (`MergeNestedIfStatement`) flattens the hierarchy of a nested `if a if b:` structure. By changing flattening it. The bodies and else statements get restructured appropriately. An example for those refactoring is as follows.

before:

```
if a:
    print(1)
    if b:
        print(2)
else:
    print(3)
```

after:

```
if a:
    print(1)
if a and b:
    print(2)
if not a:
    print(3)
```

The refactoring Add else after return, break, continue (`AddElseAfterReturnBreakContinue`) inserts an `else: ...` part after an if statement where the body ends with a `return`, `break` or an `continue`. Therefore it changes guard clauses in to if, else statements. An example for those refactoring is

before:

```
def foo():
    if a:
        return 1
    return 2
```

after:

```
def foo():
    if a:
        return 1
    else:
        return 2
```

## 2.4 Genetic Algorithm

A genetic algorithm is a meta-heuristic commonly used to generate high-quality solutions to optimization problem given resource boundaries. In such an algorithm, a population of candidate solutions is evolved toward better solutions. In this section, we aim to give understanding of our specific setup including representation of individuals(chromosome) to be evolved, the fitness values to be minimized, and operators to be applied during each generation.

Note that the below concept is implemented using DEAP[4], an evolutionary computation framework for python.

### 2.4.1 Representation

A population consists of a fixed number of individuals. An individual is created with a DEAP internal type. Each individual carries genotype, referring to the representation and a phenotype that acts as a fitness value for the genotype.

Our genotype is an ordered set of fixed length, with genes being tuples of refactoring type and target. The refactoring type is one of the 15 types defined earlier in section [2.3.2], such as `ForToWhile` or `StaticToInstance`. The target specifies the node of the AST, being identified with the object id of that node in the initially parsed tree. This is to cope with the fact that ast node's inherent ids differ in each deepcopied syntax tree.

Phenotype is a tuple of fitness values. See section [2.4.2] for further reference.

### 2.4.2 Fitness

In this research, we aim to minimize the similarity between the code base and code base after application of the respective sequence of refactorings, while also minimizing length of the refactoring sequence. For this multiobjective optimisation task we decided to use NSGA-II with the similarity score and the length of the refactorings as fitness functions. To preserve the best Individuals we decided to add an archive which stores the best paretofront in each generation.

### 2.4.3 Initialisation

We initialise the first Population with an Uniformly distributed sampling out of the applicable Refactoring types. For each selected refactoring type we then sample uniformly the application target. The sampled refactorings are stored in a sequence with variable length. This is the genetic that makes up an Individual. The population size is determined fixed and determined by a hyper parameter, currently set to 15.

### 2.4.4 Operators

- **Selection**

DEAP supports the NSGA-II selection method. We simply utilized the default selection, after checking the underlying logic of non-dominated sort and crowding distance. Given a pool of individuals and a number of selected pools, NSGA-II selects the best(in terms of fitness values) among them and returns.

- **Crossover**

Here, we implemented a custom operator of uniform crossover. In the case of crossover between individuals of different length, the bit from the longer individual had an equal chance of (i) either staying in that individual, or (ii) being removed from that individual and appended to the other(shorter) individual.

- **Mutation**

We implemented three types of mutation: ADD, SUBTRACT, REPLACE. ADD mutation adds fixed number of random genes with bias so that each refactoring type has equal chance to be selected. SUBTRACT mutation gets rid of a fixed number of randomly sampled genes. REPLACE mutation is essentially a combination of ADD and SUBTRACT mutation, by picking random genes from the chromosome and replacing it with genes selected from the available genes. Each mutation has an equal chance of  $\frac{1}{3}$  to be applied.

## 3 Results

### 3.1 Experimental Setup

#### 3.1.1 Target Codebase

We applied our tool to two target codebases, `_classical_simulator.py` from ProjectQ[5] and `pycode_similar.py` from our utilized similarity detector[2] itself.

#### 3.1.2 Hyperparameter Setting

The server response time of **MOSS** was the bottleneck of fitness evaluations, that is why we used a different experimental setting for **MOSS** and **pycode-similar**, reducing computational load in when using **MOSS** as a similarity measure.

We tuned parameters for faster convergence and better Pareto fronts by manually observing result population’s fitness values and tweaking values accordingly. Finding better hyper parameter tuning for GA remains as further work.

- Using **pycode-similar** as Similarity Checker

# of population	25
# of generations	100
crossover prob.	0.5
mutation prob.	0.7

- Using **MOSS** as Similarity Checker

# of population	15
# of generations	30
crossover prob.	0.5
mutation prob.	0.7



### 3.2 Fitness Value Distribution

During generations, we observed fitness value distribution change by logging the standard deviation, minimum, average, and maximum fitness value for each objectives among populations. We present the results obtained from the codebase `pycode_similar.py`. Although we aim to minimise both two objects (sequence length, similarity score), the result change showed that the sequence length tends to increase, to introduce more individuals with lower similarity score. We set the initial population’s sequence lengths to around relatively short, being around 15 and this tendency is strongly marked in our result. With this result, we decided to use *Archive* to eventually preserve shorter sequences with reasonably low similarity scores by preserving the individuals contained in the best Pareto fronts for each generation.

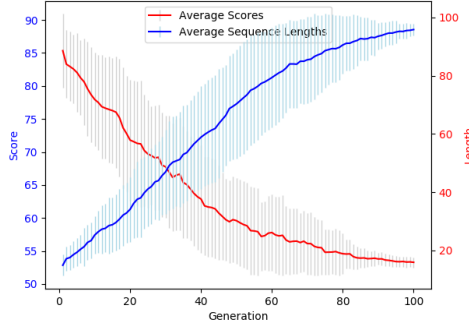


Figure 4: Average Fitness Change

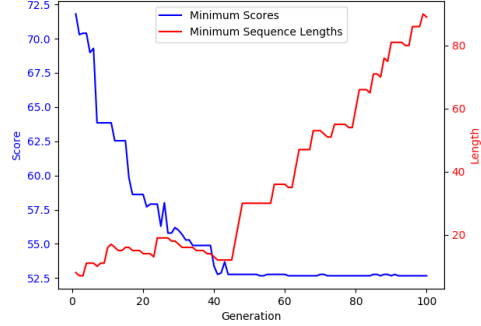


Figure 5: Minimum Fitness Change

More specifically, by observing the minimum fitness changes during generations, the sequence starts to bloat after around 40 generations, maintaining constant minimum similarity score around 52 (which means it could not succeed in finding new individuals with better score). Actually, the plagiarism score that is achievable by applying all possible refactoring operations of our tool to this codebase is 52.7693, because our tool currently identifies available refactoring types and targets by instrumenting initial source code once. To improve the implementation, we can identify more refactoring types and possible targets in the source as the codebase gets refactored, and can introduce more refactorings on refactored codebases during the evolution. However, for this feature, we have to maintain available refactoring operations set for each individual and re-instrument the changed source code which can con-

sume large amounts of memory and require a lot of additional computation. Hence, we decided to use static refactoring candidates for now.

The presented result shows the best (minimum) similarity score among all discovered individuals. Because the search space is not that big, given the limited types of refactoring operations, GA succeeds to find individuals with the best score in  $\leq 50$  generations. *The value in the parenthesis indicates length of the resulting refactoring sequence.*

codebase	similarity checker	all refactorings applied	best score
pycode-similar	MOSS	29 (97)	42 (65)
pycode-similar	pycode-similar	52.77 (97)	52.6686 (96)
projectQ	MOSS	35 (55)	35 (53)
projectQ	pycode-similar	44.10 (55)	43.86 (53)

### 3.3 Result Pareto Fronts

In the sense of finding more valid test cases for each plagiarism checker, we bring the concept of *critical plagiarism*, not only with the lower score that can bypass the detecting tools, but also having reasonable length of refactoring sequence that can be also produced by human, the malicious users (or lazy students). By multi-objectives search minimising two objectives of score and length, we can obtain more diverse individuals locating on evolving Pareto fronts. Here, we present the set of Pareto fronts observed during the evolution steps, and see whether our GA procedure actually contributes to obtain more critical plagiarism scenarios.

#### 3.3.1 Result of PyCode-similar Checker

The non-dominated sorting result of archived population on each population is as following figure 6 and figure 7. We selected best 5 fronts to show for each 10th generation. The number of individuals on the fronts are continuously increasing, showing the validity of our approach. The distribution of Pareto fronts differ from the target codebases, but we could observe some common tendency of having some especially effective refactorings that lower the score significantly without adding many operations. Also, under the certain score, it seemed to become much harder to decrease the score with a few number of refactorings. These observation would be discussed deeper on next section.

Figure 6: target codebase: `pycode_similar.py`

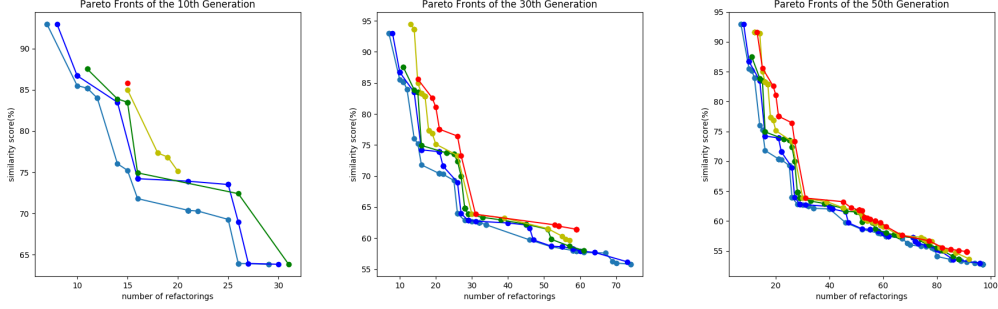
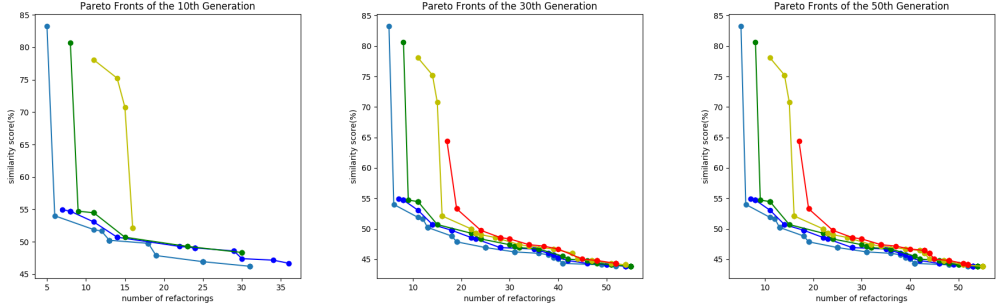


Figure 7: target codebase: `classical_simulator.py`



### 3.3.2 Result of MOSS Checker

On the other hand, the result in following figure 8 and figure 9 from MOSS checker showed more linear-shaped fronts. This may indicates that our refactoring operators affect the score more equally for MOSS than for `pycode-similar`. In common, more and more diverse individuals are introduced during generations by archiving.

### 3.3.3 Hypervolume

To ensure that our search actually guided to find more critical plagiarism cases, we measure the hypervolume of the resulting population of each generation. Both for the MOSS and `pycode-similar` detectors, the value of

Figure 8: target codebase: `pycode_similar.py`

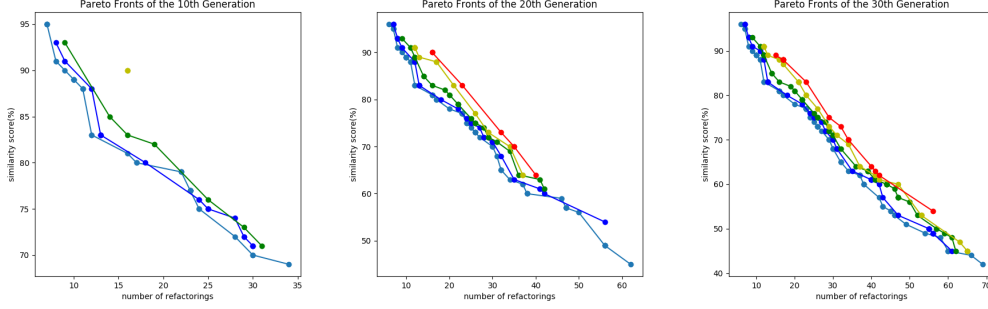
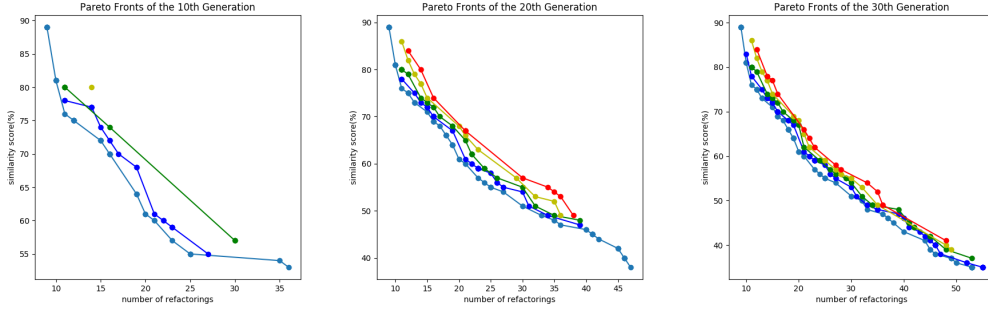


Figure 9: target codebase: `classical_simulator.py`



hypervolume showed significant improvement during generations, for example, from 2709.63 to 377.84 after 100 generations with `pycode-similar` as a detector and target codebase.

### 3.4 High-level Findings

#### 3.4.1 Difference of MOSS and `pycode-similar`

MOSS utilizes longest matching sequence for plagiarism detection[6], while `pycode-similar` utilizes normalized Python AST representation for detection. From here, we can infer that MOSS is relatively sensitive to syntax while `pycode-similar` is relatively sensitive to semantics and data structure.

These aspects can explain the stronger linearity of Pareto fronts in MOSS

compared to `pycode-similar`. As the number of refactorings on the codebase increases, the similarity score of `MOSS` proportionally decreases because of the change in syntax. On top of this, changes in semantics(data structure) are additionally considered in the case of `pycode-similar`, which additionally drops the similarity score when the number of refactorings is relatively small( $\simeq 20$ ).

### 3.4.2 Refactoring Type Strength

The strength of decreasing the similarity score had different levels as follows.

- High impact: Addition of guard clauses  
When testing with `pycode-similar`, adding `else` after `return`, `break`, `continue` significantly dropped the similarity score(around 90 down to 60) when the number of refactorings is relatively small( $\simeq 10$ ).
- Medium impact: Conditional refactorings  
Similar to the above, refactoring conditional statements(split `and/or` conditional, merge nested `if` statements) had considerable impact of lowering the similarity score. Together with the guard clauses, it can be observed that the plagiarism checker was vulnerable to branch modifications in the codebase.
- Little but consistent impact: Expression-level refactorings  
From  $\geq 40$  refactorings, expression-level refactorings were keep being added as generations evolved. These refactoring did not dramatically lower the similarity score but consistently lowered similarity as they were added more.

## 4 Future work

### 4.1 Improvement to the System

As the current iteration of `PlagGen` serves as a proof of concept, further improving it over a longer time horizon should yield more tangible results.

First, we want to explore the capabilities of refactorings and further expand our custom refactoring engine by adding more operators. As most of our refactorings operate on a low expression or conditional level, having bigger, high-level changes introduced to the overall program structure could

result in much higher score differences. Looking at a bigger range of achievable similarities available should help the GA come up with more meaningful Pareto fronts.

So far, we have only tested the refactorings by rerunning the refactored codebases (ensuring they do not get broken on a fundamental level). To ensure that the operators do not change the functionality of the programs in any way, we need to include an interface that can run the newly created codebases against a series of unit tests. We would then use this feature to rigorously test all of our proposed refactoring operators. Verification methods for proofing that the programs retain the same functionality are theoretically available, but developing such proofs in this context would be tremendously difficult.

At the moment, our tool only works with codebases consisting of single files. Extending it to work with spread out codebases enables two things. First, we can assess our tools against a wider variety of real-world codebases, as most codebases have multi-file structures. Second, the effect of moving lines of code between files on the similarity score could be interesting to observe and is definitely worth exploring.

We did not have the opportunity to spend extensive time on hyperparameter tuning for our GA. Naturally, quality of the Pareto fronts and the speed in which those are found should improve greatly when utilizing adjusted parameters when running the GA. Also, adjusting operators like our crossover or the details of our mutations could yield further improvements of the GA in all aspects.

The performance of our tool also marks an area of improvement. Especially when it comes to fitness evaluations, working around bottlenecks like the server response time of MOSS is important to realistically use these measures for our genetic algorithm. Our current plan involves parallelization of these requests to the server using multiple user access ids simultaneously. As far as the refactorings are concerned, caching and reusing already partially refactored codebases could lower the required code restructurings per fitness evaluations and result in an overall improved performance.

## 4.2 Human Readability Measure

For now, we have focused on fooling the plagiarism detector, no matter how the code might look like to a human reader. However, refactorings that lower the similarity score but make the automatic change obvious to human

onlookers might not be as critical of a change, as they would not be a threat in a real-world situation. Our algorithm should prefer hard to detect changes for both humans and machines, but to discriminate against individuals in such a way would require a way to measure how likely a piece of code was written by a human. Approaches for this have been found in NLP and it would be interesting to see if similar techniques could be applied to source code.

### 4.3 Extended Testing

For now, we looked at `pycode-similar` as a similarity scoring system and ran some evaluations against MOSS. In the future, the potential of our tool in discovering weaknesses in plagiarism software could be better assessed by testing it against a bigger variety of tools with differing codebases. Functionality to automatically benchmark and compare different plagiarism detection tools could also be desirable to make the findings more accessible for the developers of these tools.

## References

- [1] Stanford University. Moss(measure of software similarity). URL <https://theory.stanford.edu/~aiken/moss/>.
- [2] github.com/fyrestone. pycode-similar. URL <https://pypi.org/project/pycode-similar/>.
- [3] D. Clark C. Ragkhitwetsagul, J. Krinke. A comparison of code similarity analysers. *Empirical Software Engineering*, doi:10.1007/s10664-017-9564-7, 2018. URL <https://link.springer.com/article/10.1007/s10664-017-9564-7>.
- [4] Marc-Andre Gardner Marc Parizeau Christian Gagne Felix-Antoine Fortin, Francois-Michel De Rainville. Deap: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13, 2012.
- [5] ProjectQ-Framework. Projectq. URL [https://github.com/ProjectQ-Framework/ProjectQ/blob/e041ac7aac5225f339211d848d12b407e0f29f99/projectq/backends/\\_sim/\\_classical\\_simulator.py](https://github.com/ProjectQ-Framework/ProjectQ/blob/e041ac7aac5225f339211d848d12b407e0f29f99/projectq/backends/_sim/_classical_simulator.py).
- [6] A. Aiken S. Schleimer, D. Wilkerson. Winnowing: Local algorithms for document fingerprinting. *SIGMOD*, doi:10.1145/872757.872770, 2003. URL <http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf>.