Automated Program Repair

Naman Ahuja, Tanaseth Julerttrakul, Weihao Kang, Edith Kung, Owen Wang

1 INTRODUCTION

The study was conducted in order to understand how automatic program repair works. Astor – an automated program repair tool – was used to perform automatic program repairs on multiple Java defects from the QuixBugs Benchmark.

2 TOOL DESCRIPTION

Astor (Automatic Software Transformations for program repair) [2], is an open-source automation program repair framework written in Java that aims to provide the capability to repair both Java and Python programs without any human interventions by utilizing the idea of search-based software engineering (SBSE). It was developed by Inria, the University of Lille, the Université Polytechnique Hauts-de-France, and KTH Royal Institute of Technology. Astor has three major modes of repair: `jGenProg2`, `jKali`, `jMutRepair`. It can be executed as a batch program on any Java projects with specified test cases. It also allows the users to customise the parameters such as the number of generations and iterations.

For this research, we will aim to focus on repairing bugs in Java using Astor's `jGenProg2` mode. Astor is a Java implementation of genetic programming, which aims to recursively generate a population and use mutation to synthesises fixes for a particular issue. For `jGenProg2`, it first performs fault localisation by utilising a given test suite with multiple test cases to identify suspicious lines in the buggy program. Then, it attempts to apply remove, replace, or insert operations on code statements based on the particular suspicious line, and evaluate the temporary snapshot using the test suite. The primary goal is to minimise the outcome of the fitness function, which is the number of passing test cases over failing test cases. Final patches and fitness values will be generated after the termination of execution and the output can be further investigated for manual repair.

3 SELECTED BUGS

QuixBugs [1] is a program repair benchmark based on the Quixley Challenge. We obtained a set of programs, each with a one-line defect, and the corresponding test cases from QuixBugs. The list of bugs that we targeted are:

(1) Longest Increasing Subsequence (LIS) - Naman Ahuja
(2) Depth-first Search (DFS) - Tanaseth (Theo)
(3) K-Heap-Sort - Weihao Kang
(4) Merge Sort - Owen Wang
(5) Knapsack - Edith

The explanation of bugs in each program along with their expected behaviours will be elaborated in detail in the later individual sections. The bugs Pascal and Wrap were also analysed; however, only their outputs will be reported in section 4.4.

4 METHODOLOGY

4.1 Setting up

Astor is open-source and hosted on GitHub. Firstly, Astor repository was cloned onto our local devices. Specific modifications were made to the `astor.properties` file, which is located within the `/src/main/resources/` directory of the Astor repository. Specifically, the parameters 'maxGeneration' and 'population' were adjusted to 10 and 40 respectively. These configurations served as the common terminating conditions for the program repair.

Following this, the repository was compiled using the command found within the Astor documentation. The output `jar` file was then utilised for the purpose of program repair. Notably, only the `-mode jgenprog` option – which had been designed to support locality-aware repair with the aim of reducing bug-fixing time – was employed throughout this report. Given the constraints associated with the execution of the command, which is applicable only to Maven projects, a separate Maven project was created for each targeted bug. This was achieved by utilising the following command:

```
$mvn archetype:generate -DgroupId=[bugName]
    -DartifactId=[bugName]
    -DarchetypeArtifactId=maven-archetype-quickstart
    -DinteractiveMode=false$
```

Subsequently, each buggy program, along with its corresponding test suite as provided by QuixBugs, was transferred to the newly created, otherwise empty Maven project. Special care was taken to integrate the program and test suite within the Maven project structure. This integration ensured that the original program could be compiled without errors, and the test suite could be successfully executed using the `mvn test` command. Additionally, it is worth noting that version 4.12 of JUnit was used, as the syntax of the QuixBugs programs was compatible with this particular version.

For reference, the `Bugs` folder located within the main directory contains individual Maven projects with the buggy programs that were the focus of our work. Within each bug's directory, you can access the program's output and corresponding Python scripts for the purpose of analysis and graphs generated through that script to facilitate a comprehensive analysis of the output data. The output of Astor are also saved in the same directory

4.2 Executing

The Astor command given in the documentation [2] was run on each project individually. The command output was stored as a log file for interpretation. After the execution, two important files were generated for each project: astor_output.json, which stores the patches for successful fixes, and suspicious_*.json which stores the suspicious scores for suspicious lines found.

When the Astor program terminates, a folder `AstorMain-*` would be generated under the `output_astor` folder with the following structure:

```
AstorMain -*
 ├─bin
 │  └─... (binary executables)
 ├─src
 │  ├─default
 │  │  └─... (source file)
 │  └─variant-* (if patch found)
 │     ├─... (fixed version of source file)
 │     └─patch.diff
 ├─astor_output.json (summary of execution)
 └─suspicious_fix.json
```

4.3 Interpreting

After the execution was complete, the log of the algorithm was then analysed with the Python script to identify the number of successfully compiled patches, the number of operations performed, and the lines of code the operations had been performed on. The graphs were also generated for better visualisation and for us to spot any patterns that might arise.

4.4 Output

Table 1 shows the summary of the output given by Astor for multiple bugs, including all five targeted bugs, with the number of output patches, number of unique patches, and the number of *logically correct* patches, which are the patches that would be implemented if a developer were to go through and fix the bug. The logically correct patches, which we will now be referred to as *correct patches* or *correct*, were manually checked after the patches were generated by Astor.

It was found the algorithm did not generate any patches in 5 out of the 7 bugs we performed the experiment on. And for the bugs that the patches were generated, only one of them was correct. Moreover, increasing the number of maximum generation or population size did not accommodate the generation of correct patches whatsoever.

Bug Name	Total	Unique	Correct
LIS	1	1	1
DFS	79	11	0
K Heap Sort	0	0	0
Merge Sort	0	0	0
Knapsack	0	0	0
Wrap*	0	0	0
Pascal*	0	0	0

Table 1: Summary of the patch outputs for each bug. The bug names with an asterisk are not included in the individual reports.

5 CONCLUSION AND REFLECTION

JGenProg follows a systematic approach for patch generation. It starts with the fault localization method to identify suspicious lines within the program. Once these suspicious lines are pointed, JGenProg proceeds to introduce modification points around them.

The core of JGenProg's approach is the "generate-and-validate" technique. After generating a variant, it validates it using the provided test cases to determine whether the generated code represents a plausible patch. One of the most significant values it can provide is to provide some level of automation to the debugging process, which could potentially lessen the time and effort of developers.

JGenProg operates at the statement level, allowing for statements to be inserted, deleted, or replaced within the code. However, a crucial limitation due to its implementation is that it can only provide a patch if the required modification is already within the existing code from elsewhere in the same program. Consequently, JGenProg may not be able to find solutions to problems that cannot be addressed using the specified operations or are outside the program's scope which is the reason it was not able to solve most of the bugs.

An interesting aspect to note is that JGenProg also produces patches that pass the test cases but do not constitute the correct solution for the program. Therefore, it is important to verify that the generated patches are not overfitted and genuinely resolve the underlying issues.

Additionally, a surprising observation emerged during working with the tool: even when only failed test cases were provided as input, JGenProg demonstrated the ability to propose potential patches, emphasizing its adaptability and problem-solving capabilities.

1 LONGEST INCREASING SUBSEQUENCE

1.1 Bug Description

Longest increasing subsequence program gives the length of the longest subsequence whose elements are in ascending order for an array of unsorted elements.

In the original, correct code, the longest variable is updated to the maximum value(`Math.max`) between the existing longest value and `length + 1`. However, in the faulty code, this logic is missing, and the longest variable is set directly to `length + 1`. This difference in logic results in an incorrect output when the buggy code is executed.

1.2 Result

The program was provided four test cases. `JGenProg` fault localization method successfully identified 14 suspicious lines, each with suspicious value of 1. Line No 28 was among these suspicious lines, where there was an actual bug. The initial program fitness score was 4. `jGenProg` then created modification points around these suspicious lines.

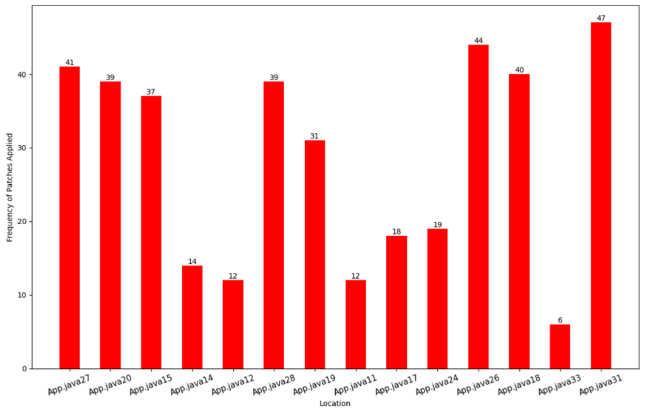


Figure 1: Number of times patches applied to suspicious lines

Figure 1 shows the frequency of patches applied to each suspicious line for each program variant. Using a population size of 40 and a maximum of 10 generations, a total of 400 program variants were generated using the operations `RemoveOp`, `InsertAfterOp`, `ReplaceOp`, and `InsertBeforeOp` and then validated against the test cases. In none of the variants, `Math.max` modification was applied because it was not possible using the above operations for this program. Upon closer examination of each iteration, modifications were done statement wise where either the statement was inserted, deleted or replaced and also the added statement was taken from existing code from elsewhere from the same program. It was observed that there were total 221 unique modifications applied and 179 duplicate modifications. Notably, all the duplicate modifications were related to `RemoveOp` operation. Figure 2 depicts the fitness values corresponding to each generated variant during the process. The graph shows intermittent breaks, indicating instances of compilation errors for certain variants. In total, there were 246 successful compilations and 153 failed compilations. Remarkably, for one variant, no mutations were left to be applied. `JGenProg`, managed to discover one potential patch for the defective program. Figure 3 refers to the diff of solution patch generated.

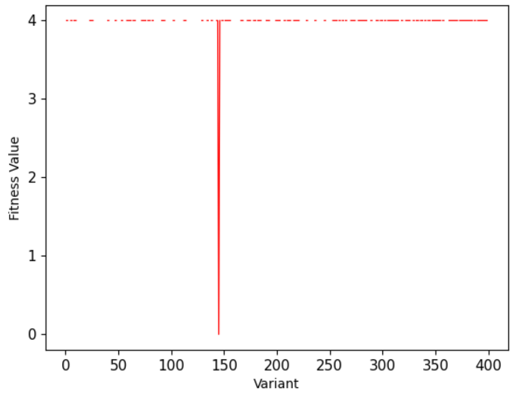
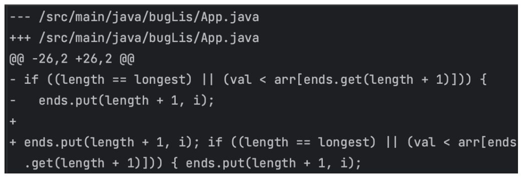


Figure 2: Fitness Values for each variant generated

1.3 Evaluation



```
--- /src/main/java/bugLis/App.java
+++ /src/main/java/bugLis/App.java
@@ -26,2 +26,2 @@
- if ((length == longest) || (val < arr[ends.get(length + 1)])) {
-   ends.put(length + 1, i);
+
+ ends.put(length + 1, i); if ((length == longest) || (val < arr[ends
  .get(length + 1)])) { ends.put(length + 1, i);
```

Figure 3: Solution Patch Diff

`JGenProg`, utilizing the `InsertBeforeOp` operation, successfully generated a patch for the program. It is noteworthy that while this patch differs from the one provided in the correct version of Quixbugs, it effectively passed all of the test cases. Also this shows that `jGenProg` picked the patch code from within the program only. To check whether the patch might be overfitted, I made a series of manual dry runs on the patched program and also provided some more random test cases. After this I was confirmed that the patch was not overfitted and indeed represented a feasible solution for the initially buggy program. An interesting observation was made during the evaluation. Only the failing test cases were provided to Astor for this program, yet it still managed to produce a potential solution.

2 DEPTH-FIRST SEARCH

2.1 Bug Description

Depth-first search is a tree or graph traversing algorithm that starts at the root node and explores each branch until reaching the leaf before backtracking. In `QuixBugs`, the function receives two input values: the root node (`startnode`) and the target node (`goalnode`). The function stops and returns true when the target node is found and returns false otherwise.

The bug in the code provided by `QuixBugs` is located in the `App.java` file. The line `nodesvisited.add(node)` is missing after line 21. This causes the program to get stuck in an infinite loop when a graph with cycles is fed into the function. Since one of the 5 test cases provided is a cyclic graph, the test suite always fails at that particular test case with the `StackOverflow` exception.

2.2 Result

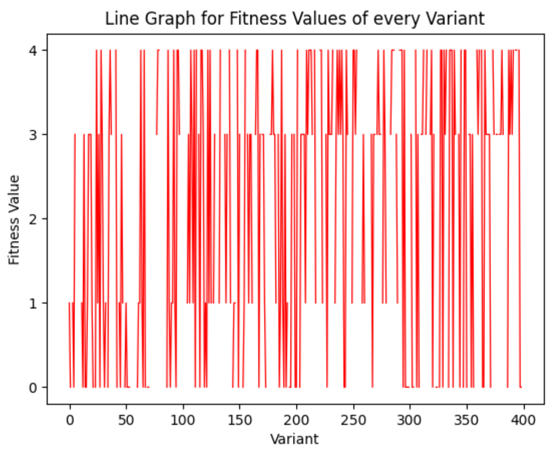


Figure 4: The graph between fitness values and the successfully compiled iterations when running Astor with the depth-first search program

In total, 79 patches were generated after running Astor; none of which targeted the `App.java` file. Fault localisation stated that line 51 in the `Node.java` file was ranked the most suspicious with the suspicious score of 1. Ranked the second was the 21st line in the `App.java` file, which was where the bug is located.

Studying the log showed that the algorithm went through 400 iterations. However, only 309 iterations could be compiled with the fitness values calculated. Figure 4 shows the graph between the fitness values and the successfully compiled iterations. Among the 400 iterations, Remove, InsertBefore, InsertAfter, and Replace operations were performed 240, 60, 44, and 19 times respectively. It is good to note that those operations only add up to 363 iterations as some of the variants in the log file were missing, and some variant didn't apply any operation at all. Upon closer inspection, it was revealed that `jGenProg` tried to fix the bug at the location mostly according to the suspicious scores, but not always. The first 4 locations where the algorithm manipulated the most times were at line 51 in the `Node.java` file, lines 20, 17, and 21 in the `App.java` file, at 110, 55, 53, and 50 times respectively. The suspicious scores of those lines were 1, 0.5, 0.447, and 0.557 respectively. Line 63 in the `App.java` file had the suspicious score of 0.5, but no operations were done on that line whatsoever.

In total, only 54 times out of 363 did `jGenProg` try to fix the bug at the correct location (at line 21 in the `App.java` file); the algorithm manipulated the code using the correct operation (`InsertAfter`) only 5 out of 53 times. Figure 5 shows how many times, and at which line and file, the algorithm tried to manipulate while it was run.

There were only a handful of unique patches; 68 of the patches were the same – changing the code in the same file, at the same line, and in the same way. Therefore, in total, the algorithm only generated 11 unique patches.

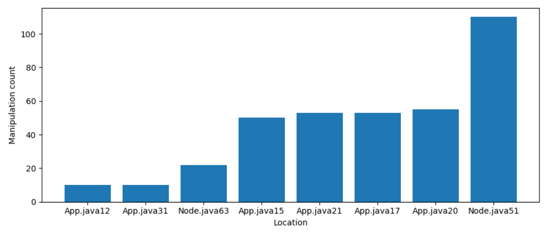


Figure 5: The graph between manipulation count and the location jGenProg tried to manipulate

2.3 Evaluation

The log showed that some of the test cases always failed when the algorithm tried to manipulate the line in the App.java file where the bug is located. Because `jGenProg` can only copy, replace, or delete lines of code, it has no chance of correctly fixing the bug as the line `nodesvisited.add(node)` isn't initially in the program.

Even though Astor gave multiple patches that could make the program pass all test cases, none of the patches were the logically correct way of fixing the bug. These patches could be regarded as overfit, since they would only pass the predefined test suite and might fail when a new test suite is provided.

3 K-HEAP-SORT

3.1 Bug Description

The `K-Heap-Sort` algorithm in QuixBug is designed to perform partial sorting using heap properties. The idea is to sort an array by a priority queue of size k. The initial operation is to add the first k elements of the array to the queue and then traverse the array. But here is where the error occurs: the original code completely traverses the entire array and does not set the correct starting point for traversal in the `for (Integer x : arr)` loop, causing the first k elements to be added repeatedly. To fix this error, start traversing from the kth element and ensure that each element is only counted once. Three of the four test cases provided by QuixBug produced ComparisonFailure errors due to this logic error.

3.2 Result

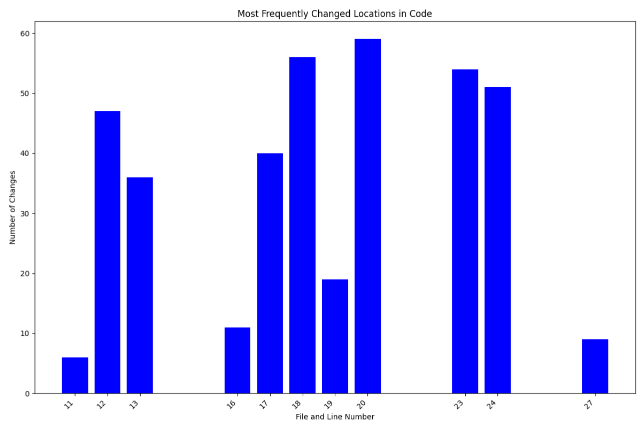


Figure 6: The graph of counting the number of frequent changes in the source code

An attempt to fix the K-Heap-Sort flaw was made using the JGenProg mode of the Astor tool, with the parameters Generation set to 10 and Population set to 40. As can be seen from the first JSON file generated (`astor_output.json`), 388 compilation attempts were made, successfully compiling 315 patches and 73 failures. Despite reaching the maximum of 10 iterations, no valid patches were found. The second JSON file (`suspicious_kheapsort.json`), the suspected defective lines, line numbers, and their probabilities are listed. The tool primarily points to lines 13 and 24 as the most likely locations of the defect with a possibility of 1. At the same time, the actual erroneous statement is located at line 17 with a probability of 0.866. This means that the tool may have yet to identify the location of the error with complete accuracy.

As indicated by the figure 6, out of the 388 attempts, the highest number of changes was made for line 20, which is `output.add(popped);`, which adds the smallest element in

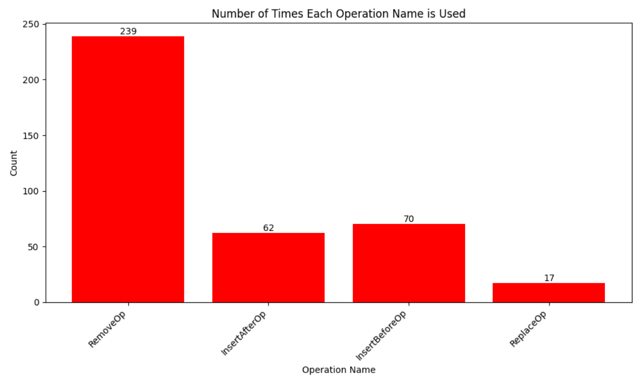


Figure 7: The graph of counting the number of frequent changes in the source code

the priority queue to the end of the `output` list, and the real error was attempted 40 times at line 17, which indicates that after several attempts at modifying the suspicious lines, these lines fail the test to get a patch.

In Figure 7, it can be seen that Astor performs a total of 388 modifications of the four operations on the source code, but the `RemoveOp` operation has the highest number of attempts at 239, and the other operations such as `InsertAfterOp` and `InsertBeforeOp` are 62 and 70 times respectively, while `ReplaceOp` has the least number of times at 17. It is clear that Jgenprog is trying to remove some lines that it thinks are wrong, attempting to remove lines 20, 12 and 18 more often, which is a redundant operation. A look at the logs shows that many more `RemoveOp` are actually invalid, and the resulting patches cannot be tested to find a solution.

3.3 Evaluation

Overall, `jgenprog` is excessively ineffective in its operations. In `for (Integer x : arr)`, it is not possible to operate on the statement inside the for loop condition, but only on the whole statement (deleting, copying, substituting), which indicates that `jgenprog` is not able to be modified for low-grained errors, no matter how much the search scope is expanded, it is still not possible to operate on low-grained errors. This means that `jgenprog` cannot work on lower granularity bugs. No matter how much the search is expanded, `jgenprog` is still unable to work on lower granularity and, therefore, does not provide a correct patch for `kheapsort` problem. If there is a problem between rows, `jgenprog` will be able to solve the problem. This is because the lowest granularity operation of `jgenprog` is the rows and not the logic in the middle of the rows.

4 MERGE SORT

4.1 Bug Description

The merge sort is a type of divide-and-conquer algorithm that splits the array into smaller subarrays, sorts these subarrays and then merges them into a single array recursively to construct a sorted array. In the buggy program provided by QuixBugs, the issue was due to an incorrect comparison expression bug in checking the array size, `arr.size() == 0`, which causes the program to always recursively merge the subarray even though there is only one element in the subarray. The correction for this bug is to change the `== 0` to `<= 1`, which will directly return the subarray if it only contains one element. A total amount of 13 test cases is provided by QuixBugs for merge sort. Initially, all the test cases result in StackOverflowError exception by running them on the buggy program, which is expected.

4.2 Result

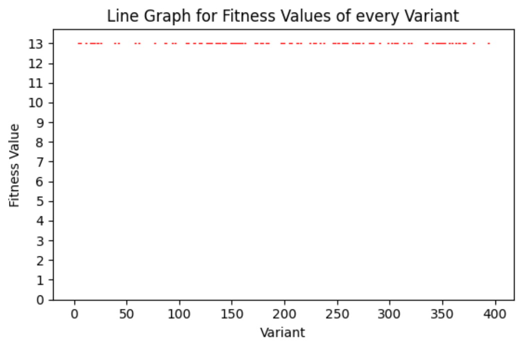


Figure 8: The fitness value for each iteration for executing on merge sort (empty slot indicates a compilation failure)

By using `jGenProg2` mode with a max generation of 10 and population of 40, no patch is output after the Astor program terminated, which indicates no solution was found. Refer to the line number of the original source code, the fault localisation mechanism outputs line 30-31 and lines 33-39 as suspicious (in `suspicious.json`), which does include the actual buggy statement in line 30. Out of all 383 iterations, as shown in Figure X, 45 attempts are modifying line 30, which is the buggy line. By analysing the runtime log, 383 iterations were performed over these lines, and among these iterations, 225 attempts resulted in successful compilations and 159 attempts caused the compilation to fail. The fitness value graph is shown in Figure 8 that indicates all of them failed to pass any of the 13 test cases after applying the candidate patch during every iteration.

Four types of operations in all patches: 73 `insertAfterOp`, 80 `InsertBeforeOp`, 204 `RemoveOp` and 26 `ReplaceOp`. We can observe the majority of the operations perform `RemoveOp`

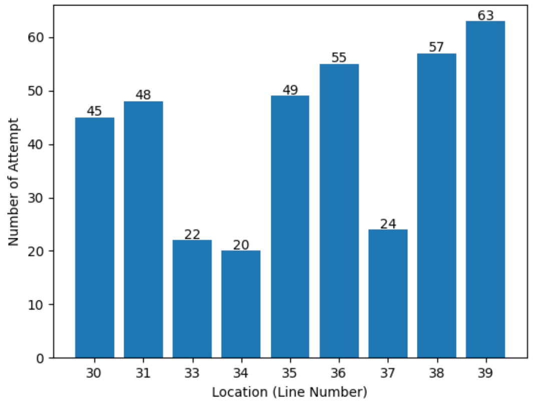


Figure 9: The number of manipulations for each line in the merge sort source code

on specious statements, which is an incorrect approach for fixing the given bug. By analysing the log in detail, several `RemoveOp` is repeated among 383 iterations, exact, removing line 36 (36 times), line 39 (37 times), if statement from line 30 to line 33 (28 times), line 35 (32 times), line 38 (32 times) and comments in line 30 (39 times). The rest of the operations are unique. Referring to the runtime log, there are also a few redundant `ReplaceOp` that attempt to replace the statement with exactly the same code, for example, one operation (child id 47) performs `ReplaceOp` on line 35 without any code changes.

4.3 Evaluation

Nevertheless, all the operations are incorrect approaches to fixing the issue in the buggy program. These operations attempt to perform delete, copy and replacement operations for the source code statements on a line-by-line basis after the source code has been parsed into abstract syntax trees in which each node is mostly a “line of code”. Exactly, regarding line 30 where the bug is located, the program always attempts to manipulate the entire if-else statement by either deleting it or replacing it entirely, without considering performing changes with the expression inside the if condition. This would be a limitation in the implementation of `jGenProg2` that always attempts to manipulate the statement level as the starting point of modification, which is too “coarse-grained” to capture the expression issue within the if condition even if we infinitely increase the search space. Therefore, we can conclude that it is expected that merge sort cannot be fixed with `jGenProg2` provided by Astor.

A potential improvement regarding this would be to reduce the granularity of manipulation. For example, instead of having a line of statement, having variables or operators could be a feasible direction. However, this would lead to an increase in complexity and computational expense.

5 KNAPSACK

5.1 Bug Description

The knapsack problem describes a situation where someone is trying to fill a fixed-size knapsack with the most valuable items. The solution provided by QuixBugs makes use of a dynamic programming approach. The function takes two inputs: the capacity of the knapsack `n`, and the list of items with their associated weights and values as a two-dimensional list of integers, and outputs the highest total value of items that have a combined weight that does not exceed the capacity of the knapsack. In the function, `memo[i][j]` refers to the highest value of items that can be selected amongst `items[0]` to `items[i]`, where the combined weights do not exceed `j`. However, the function uses a less than (<) operator instead of a less than or equals (<=) operator when comparing the calculated weights and the capacity. Hence, in reality, the program provides the output for the Knapsack problem with a capacity of `n-1`. As a result, this is considered to be a bug and fails 6 out of the 10 tests provided.

5.2 Results

`jGenProg2` makes use of fault localisation for program repair. Suspicious values were generated for 14 lines in the program, all of which have the same value of 0.77. This means that the tool is initially unable to identify the parts of the program that are more likely to contain the bug. As a result, `jGenProg2` is not able to effectively make use of fault localisation to speed up patch searching or find more patches.

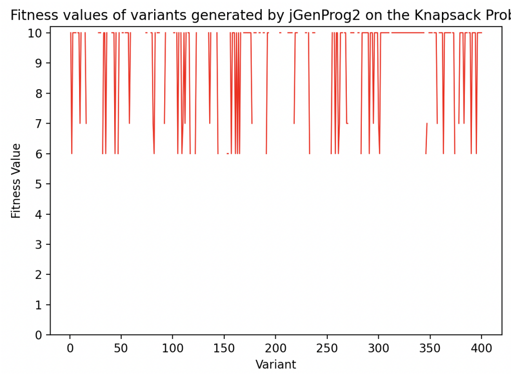


Figure 10: Fitness values of variants generated by jGenProg2 on the Knapsack Problem

As expected, `jGenProg2` was unable to find a patch for the Knapsack problem. The fitness values generated for each variant is displayed in figure 10. Out of the 400 variants, 126 did not compile successfully and hence, did not generate a fitness value, indicated by the blank spaces between the data points. It should be noted that the maximum possible fitness value is 10, meaning that the compiled program fails all 10 of the provided tests. The generated fitness values range between 6.0 to 10.0 and given that the fitness of the original program is 6.0, this suggests that `jGenProg2` did not manage to generate a solution that improves the running of the program.

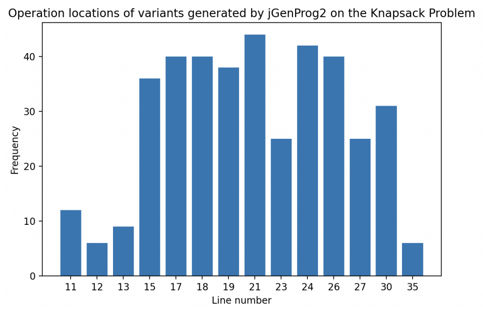


Figure 11: Operation locations of variants generated by jGenProg2 on the Knapsack Problem

Out of the 400 variants generated, 259 were `RemoveOps`, 64 were `InsertBeforeOps`, 39 were `InsertAfterOps` and 12 were `ReplaceOps`. It can be observed in figure 11 that lines 15 to 30 have been modified more often than the other lines in the program. These lines contain for loops and if statements, which are considerably more complex than the other lines that contain declaration and return statements. However, line 26, which contains the bug, did not appear to have had significantly more operations applied to it compared to other lines in the program, suggesting that `jGenProg2` was not able to identify the location of the bug either.

5.3 Evaluation

Although `jGenProg2` makes use of fault localisation techniques, this may be less effective if the generated suspicious values are very similar. This may occur due to a flaw in the design of the tests, or simply due to how the program runs. For example, if there was a test with a capacity of 0 or an empty item list, then the suspicious values would be different for certain lines, which may increase the effectiveness of `jGenProg2`.

However, as `jGenProg2` makes changes at the level of statements, it is not effective for programs whose bug cannot be fixed by removing, replacing, or inserting statements, such as the bug described in this section. Hence, even with increasing the number of generations or the population size, `jGenProg2` will never be unable to generate a patch for this program. On the other hand, Astor provides another repair mode called `jMutRepair`, which uses three kinds of mutation operators, including interchangeable operators >, >=, <, <=, ==, and `!=`. As a result, `jMutRepair`, but not `jGenProg2`, was able to find the correct patch for this bug.

6 TEAM ORGANISATION

Collaborative engagement in this project proved to be an enjoyable experience. All team members had proficiency in both Python and Java; however, our collective ease with Java made Astor the natural choice for the program repair approach.

Initially, we allocated time to establish the project setup and went through the related research papers. It is noteworthy that Owen and Weihao utilized Windows, while the rest of the group operated on macOS. Consequently, we dedicated a day to configure virtual environments to ensure uniformity in our development environments. GitHub was used for seamless collaboration throughout the project.

After the project's setup phase, we individually created custom scripts and did independent analyses. This strategy enabled us to get different perspectives on the project's output and also helped in the validation of each other's scripts and analyses. Each team member was working on a specific bug . A designated day was planned for the discussion of our findings. Python scripts were used for data analysis, and we also discussed how can we improve the scripts to get more insights from the output. Edith and Theo played instrumental roles in this endeavor.

Discord was used to conduct online discussions and meetings, with the majority occurring on our college premises. We used Overleaf as the platform for writing the report. Notably, Naman had not previously utilized this tool, and we collectively assisted him in navigating its features. Each team member made contributions to every section of the primary report. There were few clashes on the structure and few results but they were resolved through constructive debates.

With this project we not only learned more about APR but also about team management and collaboration.

REFERENCES

[1] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56, 2017.

[2] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of ISSTA*, 2016. doi:10.1145/2931037.2948705.