2. Longest Increasing Subsequence

2.1 Bug Description

This is the diff code that I got when comparing the buggy code and the correct program both provided by Quixbugs for Lis program.

```
@@ -28,7 +28,7 @@

             if (length == longest || val < arr[ends.get(length+1)]) {
                 ends.put(length+1, i);
-                longest = length + 1;
+                longest = Math.max(longest,length + 1);
             }
```

In the original, correct code, the longest variable is updated to the maximum value 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.

2.2 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.

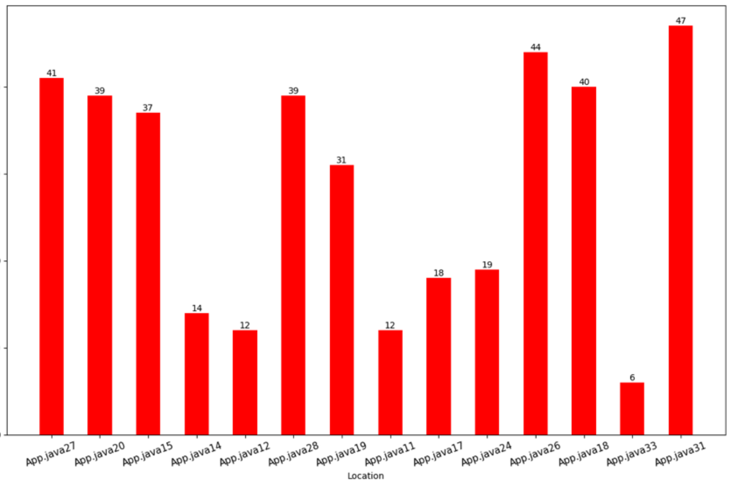

Fig1

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 were 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

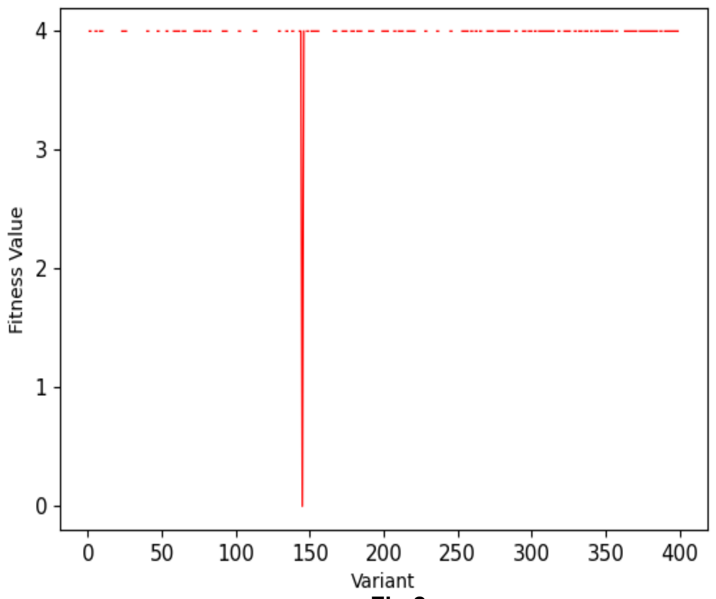

Fig 2

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 whose fitness is 0.The patch diff file generated in the output is given below.

2.3 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);
```

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.