AISTR Lab 3

Marios Marinos - 5353106
Mihai Macarie - 4668634

March 26, 2021

1 Task 1: test case generation

We used the provided test cases and have not generated cases on our own.

2 Task 2: locating bugs & Task 3: fixing bugs

As I do not think is needed to separate Task 2 & 3, I'll explain both here which will be the process that was implemented to apply automated patching.

2.1 Implementation

Step 1: Initialize a Sample based on OperatorTracker.operators to start with.

Step 2: Calculate Tarantula Score for each of these operators for the current sample and afterwards generate an initial population-based only on this sample.

Step 3: Initialize Population: The first step for a GA is the initialization of the population. What the algorithm does next is to get the sorted Tarantula scores with the respective operator index and based on this, picks the N (best_tarantula) biggest tarantula operators. We apply the roulette wheel on that samples based on their Tarantula score and return the indexes that will be mutated. How many operators will be mutated is a hyperparameter called MUTATE_OPERATORS. The mutation works as follows: pick a random operator but the one that is currently.

Step 4: Selection: Next step for a GA is how to select the samples that will pass to the next generation. The fitness function used is merely Passed Tests / Total Tests. I've implemented two different approaches here.

- The first approach (PickBest() method) is as follows: Algorithm picks the N best samples (BEST_SAMPLES) to mutate them by MUTATE_OPERATORS operators with the same approach as discussed before, and then pass them to the next generation whereas

the rest (POPULATION-BEST_SAMPLES) samples left are created by cross-overing the best samples again. (Both 1-point and 2-point crossover is implemented and can be specified by WHICH_CROSSOVER hyperparameter).

- The second approach (RoulleteSelection() method) is the same as the one before but when it comes to filling the rest samples left (POPULATION-BEST_SAMPLES) after picked the best ones, it uses again Roulette's Wheel to decide which samples will be cross-overed.

Step 5: Repeat Step 4 until the Fitness is > 0.99 or when a specific time has been reached.

2.2 Experiments

The hyper parameters that I chose based on experiments, used for the graphs and the table, displayed on 2.2 and 2.2 respectively, are as follows:

- POPULATION = 24
- BEST_SAMPLES = 8
- best_tarantula = OperatorTracker.operators.length;
- WHICH_CROSSOVER = 2-points
- MUTATE_OPERATORS = 2 (Per sample)
- RUNTIME = 30 minutes each problem from 11 to 15.

Table 1: Fitness Table

	Fitness (Approach Roulette)	**Fitness (Approach PickBest)**
Problem 11	100% (400 seconds)	100% (220 seconds)
Problem 12	∼82%	∼93%
Problem 13	∼87%	∼98%
Problem 14	∼93%	∼91%

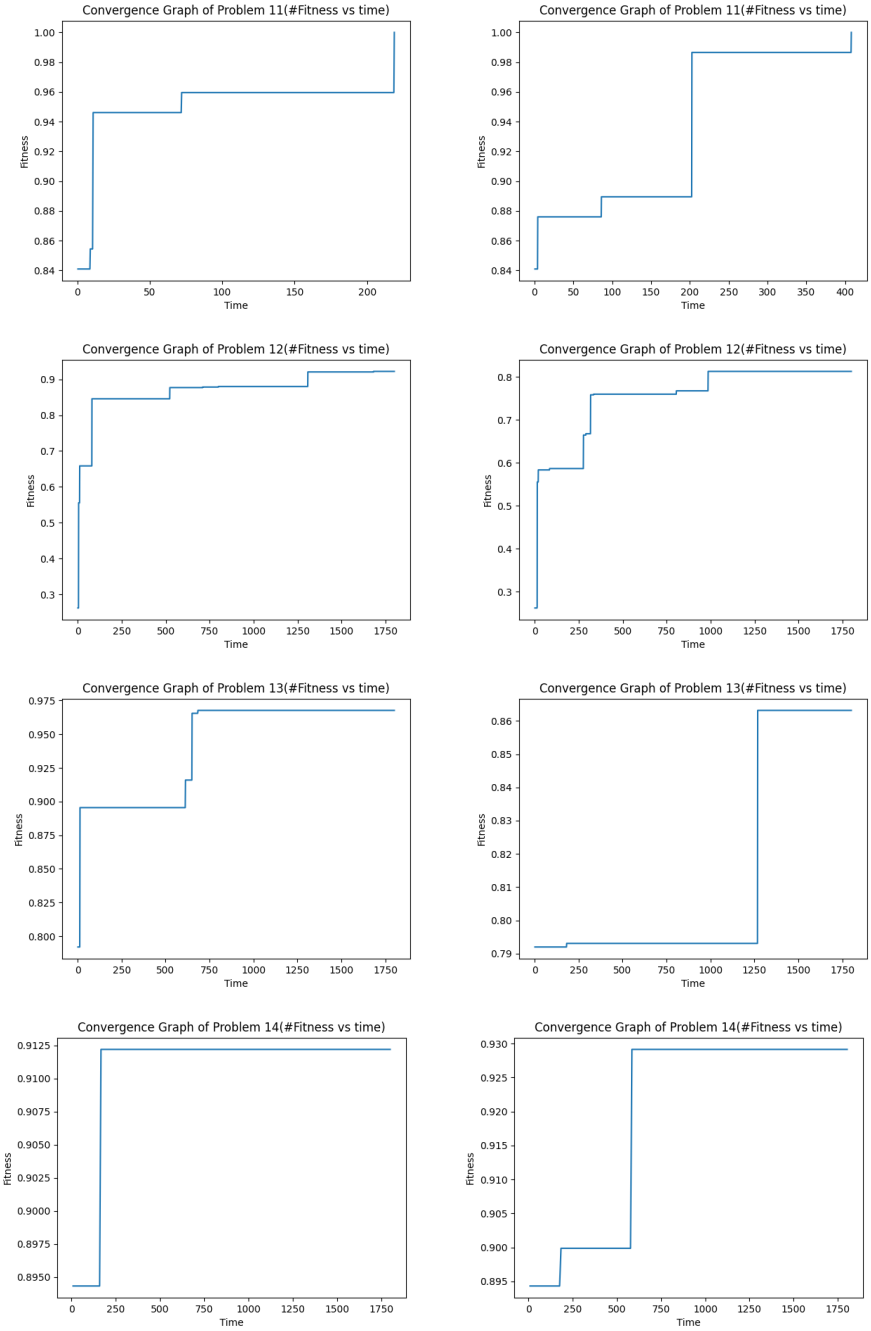

Figure 1: Number of Fitness (%) on the vertical axis, against the seconds on the horizontal axis, for problems 11 to 14 from left to right.

2.3 Improvements

Some improvements that I'd like to implement but had no time to do so would be as follows :

1. Try the Fitness Function that was proposed in the lecture (W_{pos} * #PassingTests + W_{neg} * #FailingTests).
2. Keep track of the latest modified operators, for each sample, so as not to mutate them again very early, but instead after some iterations (e.g. 20).
3. On Approach 1 & 2 I could also pick the best samples twice each instead of once from the 10 to have more mutations of the best samples.

Finally, problem 15 after 30 minutes with the same parameters didn't have any improvement which is expected as the problems get harder from 11 to 15. In Table 2.2 and the Figures 2.2 it seems that the second approach works way better than the Roulette. To me, it seems that this is expectable since the Roulette doesn't pick always the best samples but some times getting random worse samples and cross-over them.

3 Task 4: Genetic Programming using ASTOR

For this task, we analyzed all of the "buggy" RERS problems using ASTOR. ASTOR was used for all the problems with the following parameters, according to the ASTOR README page and course guide:[1]

```
java -cp \$(cat /tmp/astor-classpath.txt):target/classes
    fr.inria.main.evolution.AstorMain -mode jgenprog -
    srcjavafolder /src/main/java/ -srctestfolder /src/test
    /java/ -binjavafolder /target/classes/ -bintestfolder
      /target/test-classes/ -location /home/str/buggy/
    Problem5\_buggy/ -maxtime 5 -maxgen 3000 -scope
    local -population 8 -operatorspace relational-Logical-
    op
```

The results are shown in Table 2, where the running time was generated using **time** command - real timing value.

For all the problems, we manually analyzed the patches generated by ASTOR. We define a "meaningful patch" as a patch that is identical to the original RERS problem or logically congruent. For example, if we have the condition $a == b$, then a patch with $a == b$ or $!(a! = b)$ would be accepted. These results are also in Table 2. When we talk about the problems marked with *****, ASTOR has not created any patches using the parameters above. Therefore, we rerun these problems without the **-maxtime** parameter, which helped ASTOR finding more patches. However, for some problems, it takes a lot of time to (re)run ASTOR. Therefore, we do not have enough results for problems marked with ****** and *******.

	Running time – real [min]	No. of patches found	Meaningful patches
Problem 1	6,71	16	5
Problem 2*	121,89	6	0
Problem 3*	340,68	2	0
Problem 5**	1405,72	0	0
Problem 6***	N/A	N/A	N/A
Problem 8*	931,85	1	1
Problem 11	5,78	38	6
Problem 12	13,73	4	0
Problem 13	12,74	1	0
Problem 14	14,32	5	0
Problem 17*	296,03	3	0

Table 2: Results after running ASTOR on "buggy" RERS problems. * - problems which indicated no patches when running with **-maxtime** parameter, but with a better result afterwards. ** - problems which indicated no patches when running with **-maxtime** parameter, but we ran out of time to run it again. *** - it was taking more than 24 hours to run ASTOR and we ran of time

By analysing the data from Table 2, we think that genetic programming might be not that feasible because it does not have access to the original correct code. It can make the software working, but using patched code in production without a manual review may lead to unexpected behaviour. It might also depend on what test cases are used. On the other hand, on our implementation seemed to work quite well for the time given combined with tarantula. Finally, a possible problem might be on defining the meaningful patches and we should do that in a different way. In addition to that, it might be that the problems are quite hard, and astor is created for any kind of problems and not specific this type of problems compared to our implementation.

References

[1] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of ISSTA*, 2016.