Knowledge Search & Extraction
Project 02: Python Test Generator

Claudio Maggioni

Section 1 - Instrumentation

The script *instrument.py* in the main directory of the project performs instrumentation to replace each condition node in the Python files present benchmark suite with a call to `evaluate_condition`, which will preserve program behaviour but as a side effect will compute and store condition distance for each traversed branch.

Table 1 summarizes the number of Python files, function definition (*FunctionDef*) nodes, and comparison nodes (*Compare* nodes not in an `assert` or `return` statement) found by the instrumentation script.

Type	**Number**
Python Files	10
Function Nodes	12
Comparison Nodes	44

Table 1: Count of files and nodes found.

Section 2: Fuzzer test generator

The script *fuzzer.py* loads the instrumented benchmark suite and generates tests at random to maximize branch coverage.

The implementation submitted with this report slightly improves on the specification required as it is able to deal with an arbitrary number of function parameters, which must be type-hinted as either `str` or `int`. The fuzzing process generates a pool of 1000 test case inputs according to the function signature, using randomly generated integers $\in [-1000, 1000]$, and randomly generated string of length $\in [0, 10]$ with ASCII characters with code $\in [32, 127]$. Note that test cases generated in the pool may not satisfy the preconditions (i.e. the `assert` statements on the inputs) for the given function.

250 test cases are extracted from the pool following this procedure. With equal probabilities (each with $p = 1/3$):

- The extracted test case may be kept as-is;
- The extracted test case may be randomly mutated using the *mutate* function. An argument will be chosen at random, and if of type `str` a random position in the string will be replaced with a random character. If the argument is of type `int`, a random value $\in [-10, 10]$ will be added to the argument. If the resulting test case is not present in the pool, it will be added to the pool;

- The extracted test case may be randomly combined with another randomly extracted test using the *crossover* function. The function will choose at random an argument, and if of type `int` it will swap the values assigned to the two tests. If the argument is of type `str`, the strings from the two test cases will be split in two substrings at random and they will be joined by combining the "head" substring from one test case with the "tail" substring from the other. If the two resulting test cases are new, they will be added to the pool.

If the resulting test case (or test cases) satisfy the function precondition, and if their execution covers branches that have not been covered by other test cases, they will be added to the test suite. The resulting test suite is then saved as a *unittest* file, comprising of one test class per function present in the benchmark test file.

Section 3: Genetic Algorithm test generator

The script *genetic.py* loads the instrumented benchmark suite and generates tests using a genetic algorithm to maximize branch coverage and minimize distance to condition boundary values.

The genetic algorithm is implemented via the library *deap* using the *eaSimple* procedure. The algorithm is initialized with 200 individuals extracted from a pool generated in the same way as the previous section. The algorithm runs for 20 generations, and it implements the *mate* and *mutate* operators using the *crossover* and *mutate* functions respectively as described in the previous section.

The fitness function used returns a value of ∞ if the test case does not satisfy the function precondition, a value of 1000000 if the test case does not cover any new branches, or the sum of normalized $(1/(x+1))$ sum of distances for branches that are not yet covered by other test cases. A penalty of 2 is summed to the fitness value for every branch that is already covered. The fitness function is minimized by the genetic algorithm.

The genetic algorithm is ran 10 times. At the end of each execution the best individuals (sorted by increasing fitness) are selected if they cover at least one branch that has not been covered. This is the only point in the procedure where the set of covered branches is updated[1].

Section 4: Statistical comparison of test generators

To compare the performance of the fuzzer and the genetic algorithm, the mutation testing tool *mut.py* has been used to measure how robust the generated test suites hard. Both implementations have been executed for 10 times using different RNG seeds each time, and a statistical comparison of the resulting mutation score distributions has been performed to determine when one generation method is statistically more performant than the other.

Figure 1 shows a boxplot of the mutation score distributions for each file in the benchmark suite, while figure 2 shows the mean mutation scores.

To perform a statistical comparison, the Wilcoxon paired test has been used with a p-value threshold of 0.05 has been used to check if there is there is a statistical difference between the distributions. Moreover, the Cohen's d effect-size has been used to measure the significance of the difference. Results of the statistical analysis are shown in table 2.

Only 3 benchmark files out of 10 make the two script have statistical different performance. They are namely *check_armstrong*, *rabin_karp* and *anagram_check*. The first two show that the genetic algorithm performs significantly better than the fuzzer, while the last file shows the opposite.

[1] This differs from the reference implementation of `sb_cgi_decode.py`, which performs the update directly in the fitness function.

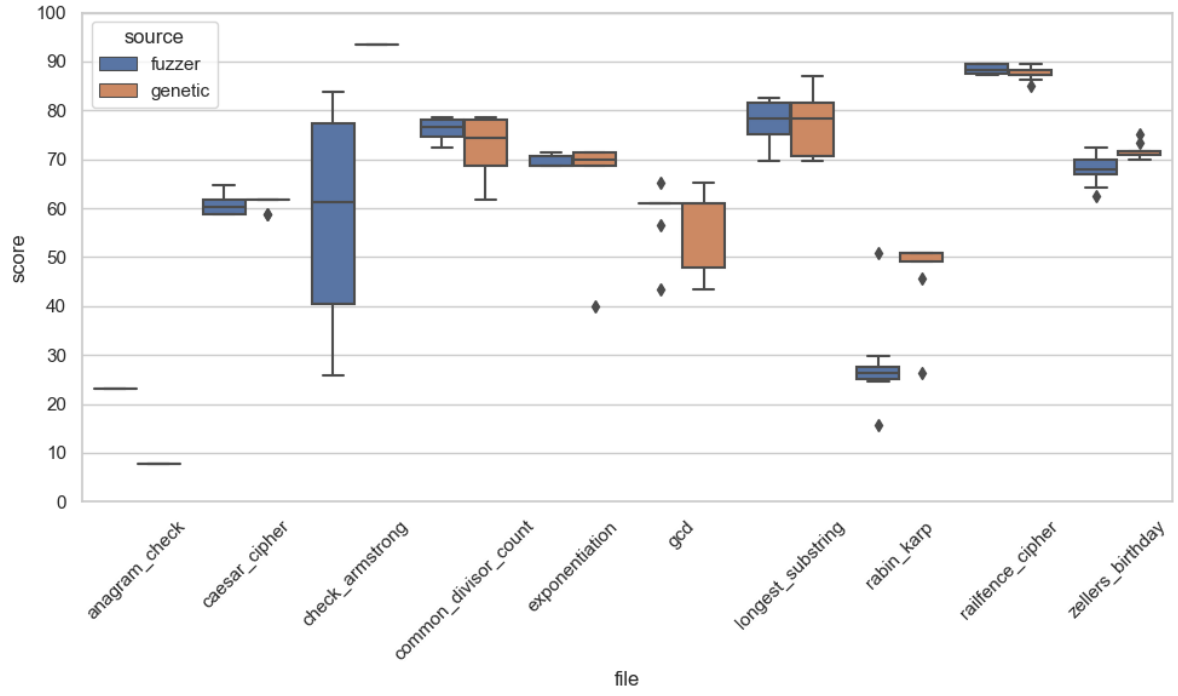

Figure 1: Distributions of *mut.py* mutation scores over the generated benchmark tests suites using the fuzzer and the genetic algorithm.

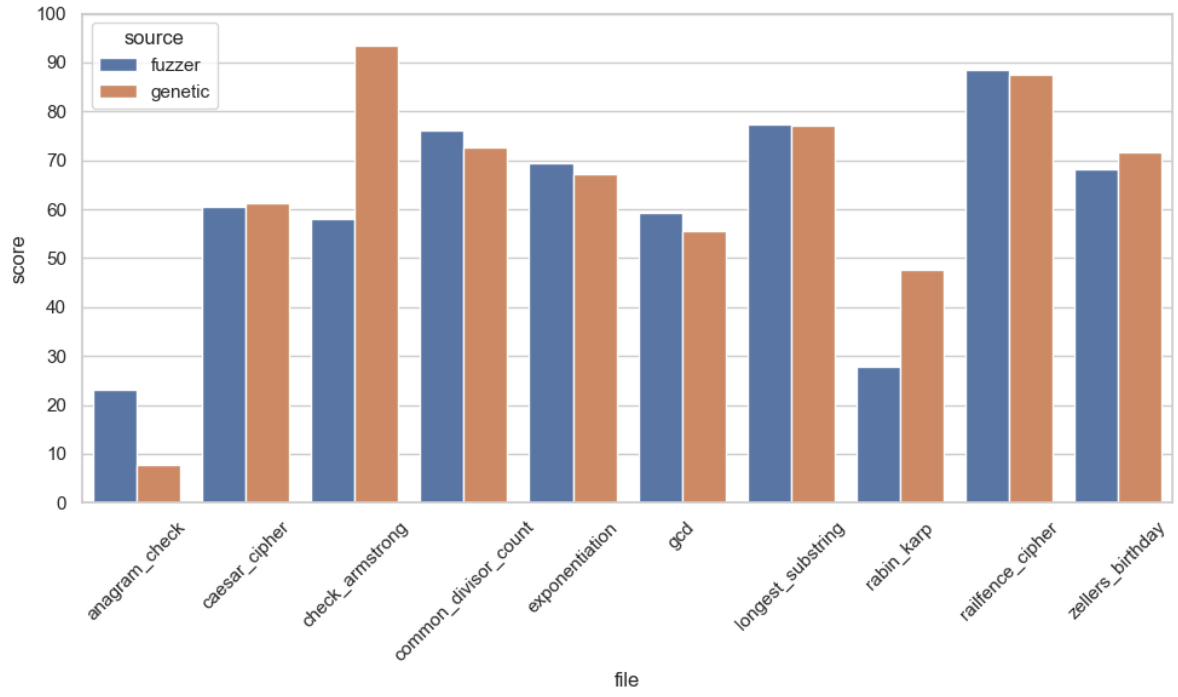

Figure 2: *mut.py* Mutation score average over the generated benchmark tests suites using the fuzzer and the genetic algorithm.

File	E(**Fuzzer**)	E(**Genetic**)	**Cohen's** $\lvert d \rvert$		**Wilcoxon** p
check_armstrong	58.07	93.50	2.0757	Huge	0.0020
railfence_cipher	88.41	87.44	0.8844	Very large	0.1011
longest_substring	77.41	76.98	0.0771	Small	0.7589
common_divisor_count	76.17	72.76	0.7471	Large	0.1258
zellers_birthday	68.09	71.75	1.4701	Huge	0.0039
exponentiation	69.44	67.14	0.3342	Medium	0.7108
caesar_cipher	60.59	61.20	0.3549	Medium	0.2955
gcd	59.15	55.66	0.5016	Large	0.1627
rabin_karp	27.90	47.55	2.3688	Huge	0.0078
anagram_check	23.10	7.70	∞	Huge	0.0020

Table 2: Statistical comparison between fuzzer and genetic algorithm test case generation in terms of mutation score as reported by *mut.py* over 10 runs, sorted by genetic mutation score. The table reports run means, the wilcoxon paired test p-value and the Cohen's d effect size for each file in the benchmark.