Knowledge Search & Extraction
Project 02: Python Test Generator

Navdeep Singh Bedi

Section 0 - Virtual Environment

Create a virtual environment using Python 3.7 using pyenv. Enter the virtual enviroment and install the dependencies using 'pip install -r requirements.txt'

Section 1 - Instrumentation

In Part01 of the project, we crawled through the whole benchmark directory using os.walk and looked for the files with the .py extension and then appended those files into a list. Then we iterated through the list one by one and made a list of function names in each python file. Then we create the AST (Abstract Syntax Tree) using the ast.NodeTransformer method of the AST module Python in the 'BranchTransformer' class. Finally, we convert python files in the benchmark folder into the instrumented version of the files named 'instrumented_filename' according to the rules given in the Part05 slides of the python by iterating to the function list and calling each function globally and also calculate the total number of python files, function nodes and comparision nodes in the benchmark directory. The results are shown in Table 1.

Type	Number
Python files	10
Function Nodes	14
Comparison Nodes	54

Table 1: Count of files and nodes found.

The code is present in 'instrumentor.py' file. The code can be run through going into the directory: project-02-python-test-generator-navdeeps350 and running 'python3 instrumentor.py'. The resultant instrumented files are named 'instrumented_filename'.

Section 2: Fuzzer test generator

In Part02 of the project the we first ask the user to enter the name of the file for which the user wants to generate test cases using the fuzzer. Then through BranchTransformer class which we created in the previous section we find the list of type of arguments we need to pass into them, then if the function take

in the arguments of type int then we ask the user to enter the minimum and maximum value of the integer, if it takes the arguments of type string we ask the user to enter the maximum length of the string, or if the function take the arguments of type int and str both then we ask to enter the maximum length of the string and minimum and maximum value of the integer and also we ask the user to enter the pool size which is the total number of generated inputs of the particular type. Then we ask the user to enter the number of test cases it wants to run this experiment for. This value will be used in the last part where we keep only the test cases that increase the condition coverage. Next according to the parameters user entered we create a data pool of particular type of input according to the function and then with 1/3 probability we select either randomly choose a input out of the pool, or do a mutation of the randomly chosen input or randomly choose 2 inputs from the pool and then perform crossover over them and choose one of the input randomly. We do this the number of test cases the user wants to run this experiment for and each time we have the dictionary of condition coverage of nodes with distances for the present which we import from 'instrumentor.py' file whoes evaluate condition functions are called by 'instrumented_filename' file when we call it from the present file and the previous experiment and for all the iterations we make an 'out[in]' dictionary which only keeps track of the inputs and corresponding outputs which led to the increase in the condition coverage. Finally, using the out[in] dictionary we generate the test case file for the given file as per the rules given in Part06 of the slides named 'test_filename'. The code is in 'testgen_random.py' file. The code can be run through going into the directory: project-02-python-test-generator-navdeeps350 and running 'python3 testgen_random.py' and entering the paramaters The resultant test files are named 'tests_filename'.

Section 3: Genetic Algorithm test generator

In Part03 of the project we use 'Deap' library to define the genetic algorithm. So first, we ask the user to enter the filename for which it wants to generate test cases. Then through BranchTransformer class which we create in the Part01 we find the list of type of arguments we need to pass into them, then if the function take in the arguments of type int then we ask the user to enter the minimum and maximum value of the integer, if it takes the arguments of type string we ask the user to enter the maximum length of the string, or if the function take the arguments of type int and str both then we ask to enter the maximum length of the string and minimum and maximum value of the integer. Then the user is asked to enter the population size, the generation size, mutation probability, crossover probability, tournament size and the number of times we need to run the experiment. Rest the function to initialize the input in case of integer inputs, string inputs and the tuple inputs with string and integer remains similar with minor changes as in the previous part. The mutation and crossover functions also remain similar for all the inputs. Finally we take the node coverage and the test case and write the files 'deap_tests_filename' for the file user has entered and generate the test case files. The code can be found in 'deap_config.py' file.

The code can be run through going into the directory: project-02-python-test-generator-navdeeps350 and running 'python3 deap_config.py' and entering the paramaters. The resultant test files are named 'deap_tests_filename'.

Section 4: Statistical comparison of test generators

Next we use the tool MutPy to inject artificial faults (mutations) into the benchmark functions under test and evaluate the fault detection capability of the genetic algorithm, considering the random fuzzer as baseline. The results for all the files are given below. The code is given in 'mutation.py' file.
The code can be run through going into the directory: project-02-python-test-generator-navdeeps350 and running 'python3 mutation.py' and entering the paramaters as in the Past02 (Fuzzer) and Part03 (GA) as this file calls the files made and created in the previous 2 sections to calculate mutation scores with the help of MutPy.
To do Wilcoxon statistical test, exit the virtual enviroment and just go in the 'Wilcoxon.py' copy and paste the mutation scores for the file you need to do the statistical testing and run the file. This is done in another file as 'Scipy' library was not working with the present virtual enviroment with Python 3.7 which is used for the rest of the project especially because of Deap library. The plot for anagram_check.py file is given in Fig. 1

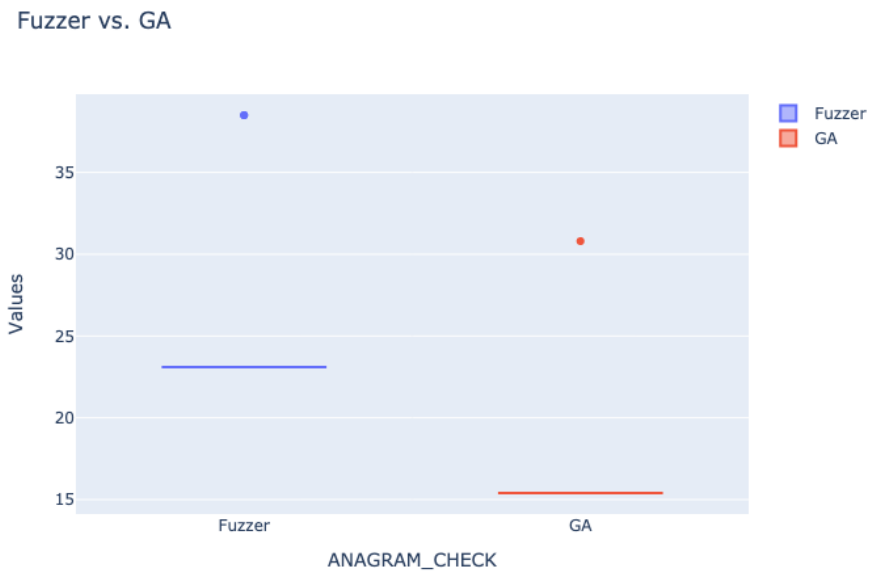

Figure 1: anagram check plot.

Fuzzer avg. mutation score: 26.18
GA avg. mutation score: 16.94
Cohen's d: 1.609
Wilcoxon's p-value: 0.003
Considering significance value as 0.05, the difference between 2 mutation scores is statistically significant.
The plot for caesar_cipher.py file is given in Fig. 2

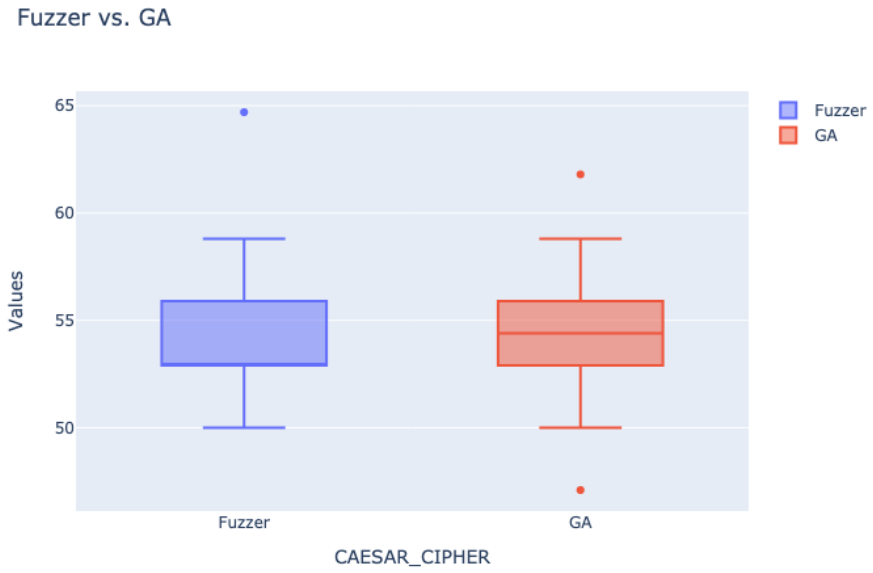

Figure 2: caesar cipher plot.

Fuzzer avg. mutation score: 54.39
GA avg. mutation score: 54.410000000000004
Cohen's d: -0.004
Wilcoxon's p-value: 0.396
Considering significance value as 0.05, there is no significant difference between 2 mutation scores.

The plot for check_armstrong.py file is given in Fig. 3

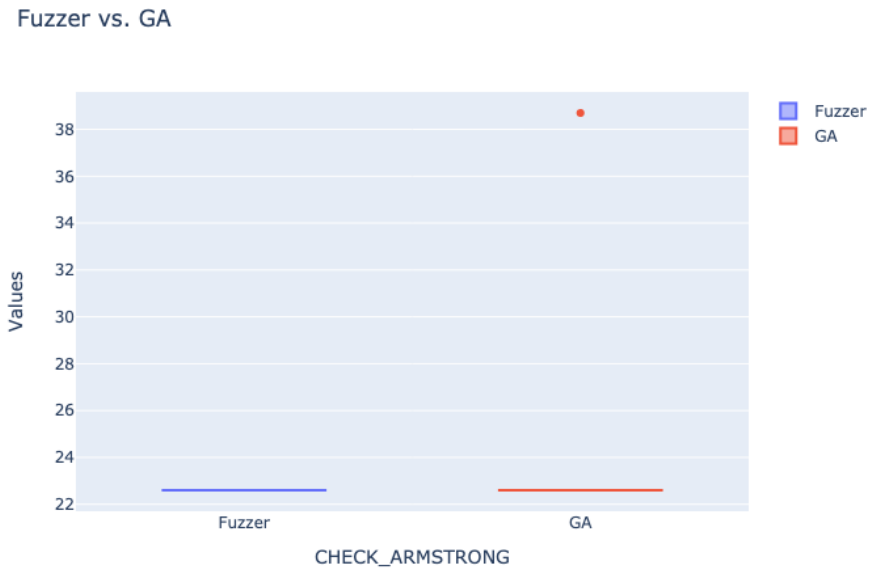

Figure 3: check armstrong plot.

Fuzzer avg. mutation score: 22.6
GA avg. mutation score: 24.21
Cohen's d: -0.447
Wilcoxon's p-value: 0.317
Considering significance value as 0.05, there is no significant difference between 2 mutation scores.

The plot for common_divisor_count.py file is given in Fig. 4

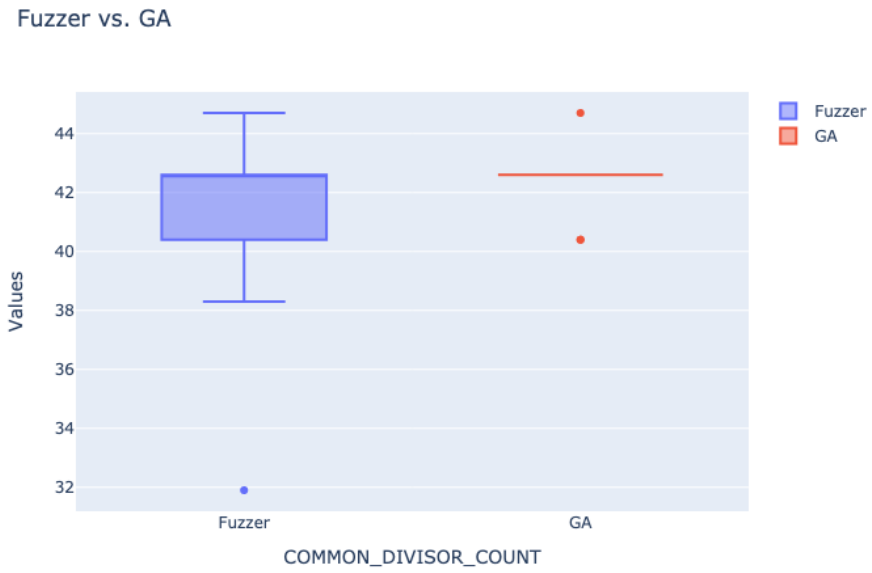

Figure 4: common divisor count plot.

Fuzzer avg. mutation score: 40.87
GA avg. mutation score: 42.37
Cohen's d: -0.556
Wilcoxon's p-value: 0.104
Considering significance value as 0.05, there is no significant difference between 2 mutation scores.

The plot for exponentiation.py file is given in Fig. 5

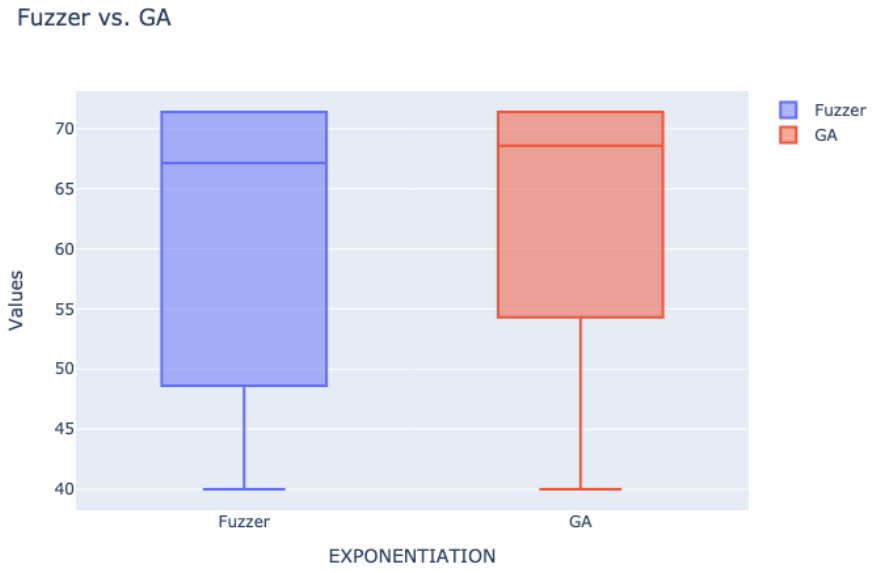

Figure 5: exponentiation plot.

Fuzzer avg. mutation score: 61.14
GA avg. mutation score: 62.29
Cohen's d: -0.096
Wilcoxon's p-value: 0.634
Considering significance value as 0.05, there is no significant difference between 2 mutation scores.

The plot for gcd.py file is given in Fig. 6

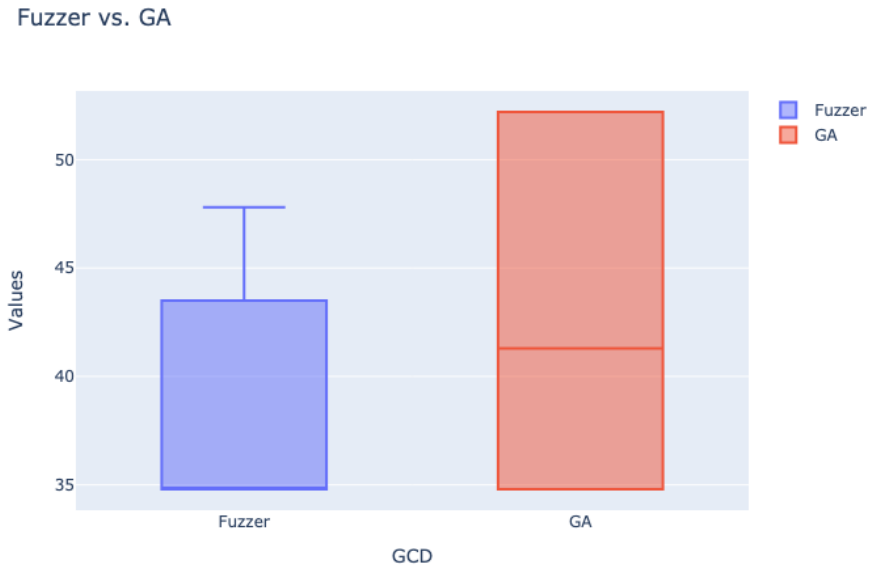

Figure 6: gcd plot.

Fuzzer avg. mutation score: 37.83
GA avg. mutation score: 42.62
Cohen's d: -0.690
Wilcoxon's p-value: 0.110
Considering significance value as 0.05, there is no significant difference between 2 mutation scores.

The plot for longest_substring.py file is given in Fig. 7

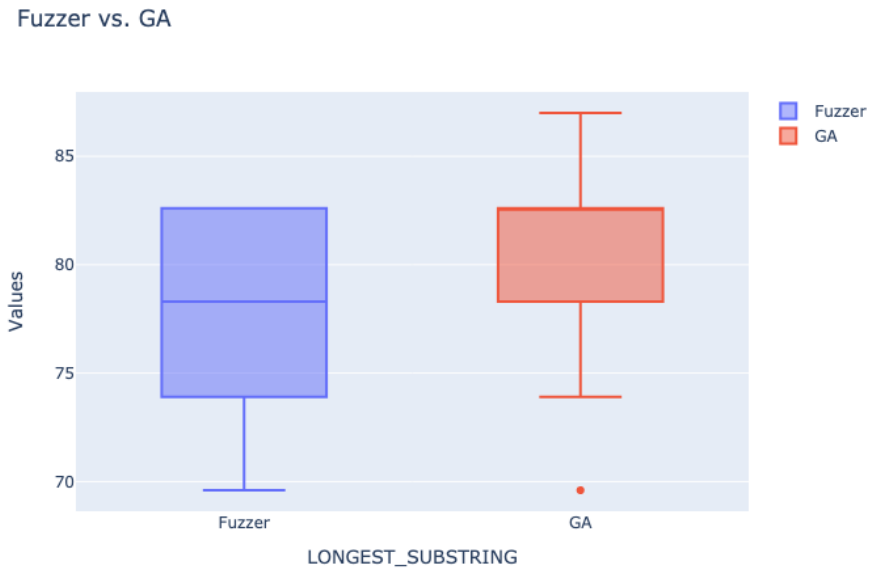

Figure 7: longest substring plot.

Fuzzer avg. mutation score: 77.84
GA avg. mutation score: 80.44
Cohen's d: -0.526
Wilcoxon's p-value: 0.397
Considering significance value as 0.05, there is no significant difference between 2 mutation scores.

The plot for rabin_karp.py file is given in Fig. 8

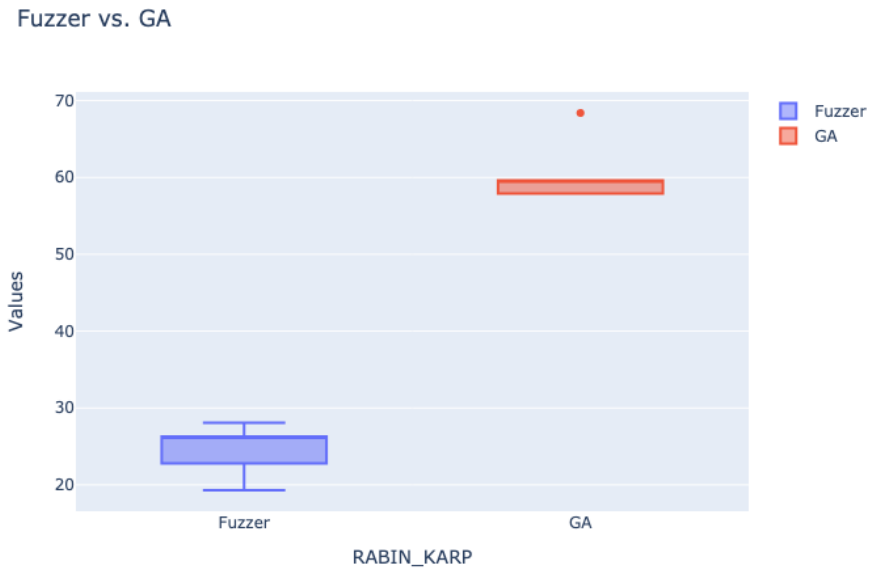

Figure 8: rabin karp plot.

Fuzzer avg. mutation score: 24.73
GA avg. mutation score: 59.96
Cohen's d: -12.245
Wilcoxon's p-value: 0.001
Considering significance value as 0.05, the difference between 2 mutation scores is statistically significant.

The plot for railfence_cipher.py file is given in Fig. 9

Figure 9: raifence cipher plot.

Fuzzer avg. mutation score: 80.96
GA avg. mutation score: 54.23
Cohen's d: 4.663
Wilcoxon's p-value: 0.001
Considering significance value as 0.05, the difference between 2 mutation scores is statistically significant.

The plot for zellers_birthday.py file is given in Fig. 10

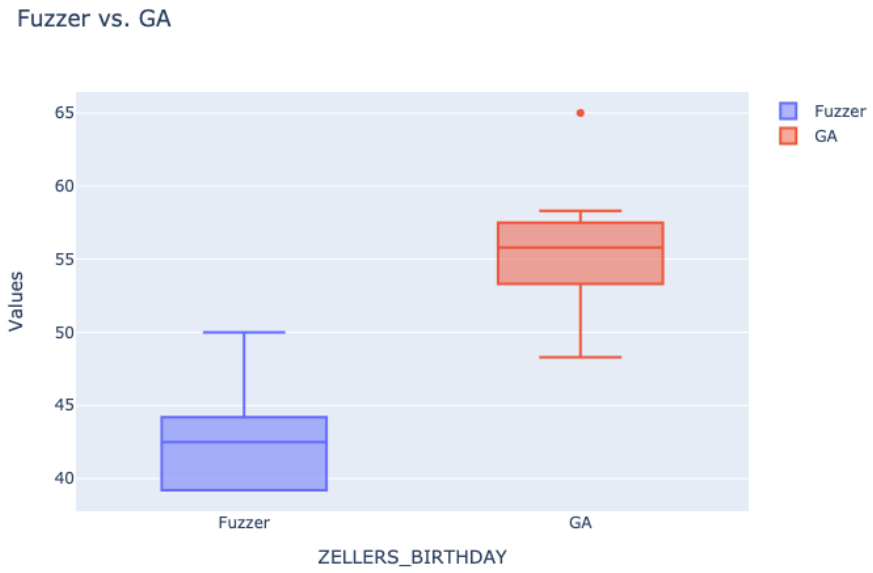

Figure 10: zellers birthday plot.

Fuzzer avg. mutation score: 42.84
GA avg. mutation score: 55.65
Cohen's d: -3.030
Wilcoxon's p-value: 0.001
Considering significance value as 0.05, the difference between 2 mutation scores is statistically significant.