Knowledge Search & Extraction
Project 02: Python Test Generator

Jury Andrea D'Onofrio

Section 0 - GitHub link

- GitHub
- commit id: 47a6456e814dcdc229c8cc1ab249e87e57241277

Section 1 - Instrumentation

The Python file `instrumentor.py` aims to implement a source code instrumentation tool to collect branch coverage information by modifying existing Python files. The script uses the tree (`AST`) module to traverse and manipulate the original source codes. The `Transformer` class extends the `ast.NodeTransformer` and inserts instrumentation code into the original source.

The code transforms function definitions by adding `_instrumented` to their names and adds calls to an `evaluate_condition` function within the Compare nodes. This `evaluate_condition` function calculates distances between operands based on comparison operators and updates dictionaries (`distance_dict_true` and `distance_dict_false`) that store the distances for each branch. The script takes in input a set of Python files in the `benchmark` folder and creates their instrumented versions in the `parsed` directory for subsequent branch coverage analysis.

In the following Table 1, you can find my results regarding the number of Python files, Function Nodes, and Comparison Nodes:

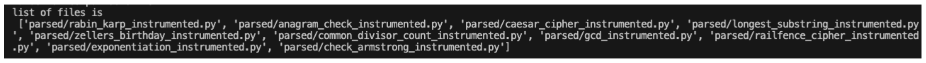
Type	Number
Python files	10
Function Nodes	12
Comparison Nodes	54

Table 1: Count of files and nodes found.

Section 2: Fuzzer test generator

This section aims to describe the steps to generate the test cases using Fuzzer. The code section follows the next logic.

Parsed Paths Function
This function has the goal to create a list containing the parsed file paths. It iterates through the files in the `parsed` folder using the `os` module. For each file, it constructs the path and if it is a file, then it adds it to the list called `list_of_files`. At the end, the function just returns the list.



```
list of files is
 ['parsed/rabin_karp_instrumented.py', 'parsed/anagram_check_instrumented.py', 'parsed/caesar_cipher_instrumented.py', 'parsed/longest_substring_instrumented.py
', 'parsed/zellers_birthday_instrumented.py', 'parsed/common_divisor_count_instrumented.py', 'parsed/gcd_instrumented.py', 'parsed/railfence_cipher_instrumented
.py', 'parsed/exponentiation_instrumented.py', 'parsed/check_armstrong_instrumented.py']
```

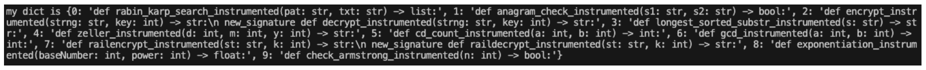
Figure 1: Parsed Paths Function result

Parsed Codes Function
This function aims to create a dictionary where the keys are the paths created in the previous function, while the values are the corresponding Python code. Moreover, it creates a list where each element is the code of a parsed file. The two data structures are then returned.

The outcomes of the described functions are then used to create a dictionary of length `list_of_files` where for each key (integer value), it stores the code of the parsed Python file. This dictionary is used to get the signatures of the methods.

Parsed Signatures Function
This function has the intent to extract the function signatures from the pre-parsed code stored in the dictionary described before. It iterates through the dictionary, identifying lines that start with the `def` keyword, indicating the definition of a function. When it encounters a function signature, the function adds the tag `new_signature` if it is the first one; otherwise, it appends the signature as it is. The extracted signatures are stored in the corresponding index of the dictionary. At the end, the function returns the updated dictionary.

After the parsed signatures function, the dictionary acts as a catalog for function signatures extracted from parsed code. Each key represents a unique file index, while the associated values are lists containing function signatures.

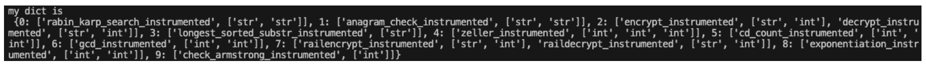


```
my dict is {0: 'def rabin_karp_search_instrumented(pat: str, txt: str) -> list:', 1: 'def anagram_check_instrumented(s1: str, s2: str) -> bool:', 2: 'def encrypt_instr
umented(strng: str, key: int) -> str:\n new_signature def decrypt_instrumented(strng: str, key: int) -> str:', 3: 'def longest_sorted_substr_instrumented(s: str) -> st
r:', 4: 'def zeller_instrumented(d: int, m: int, y: int) -> str:', 5: 'def cd_count_instrumented(a: int, b: int) -> int:', 6: 'def gcd_instrumented(a: int, b: int) ->
int:', 7: 'def railencrypt_instrumented(st: str, k: int) -> str:\n new_signature def raildecrypt_instrumented(st: str, k: int) -> str:', 8: 'def exponentiation_instrum
ented(baseNumber: int, power: int) -> float:', 9: 'def check_armstrong_instrumented(n: int) -> bool:'}
```

Figure 2: Parsed Signatures Function result

Get Types of Parameters Function
The purpose of this function is to extract information about the parameters accepted by individual functions in the parsed Python code. It does this by iterating through each file in the provided dictionary, isolating the function signatures. Then, for each of them, it extracts the sub-string between the parentheses to obtain a string in the format `name: type, ...`. This string is then further elaborated by splitting it by commas to yield an array of the form `['name: type', ...]`. The function extracts the `type` information by iterating over the array, splitting by a colon, and capturing only the second element. This information is stored as a tuple that associates each function signature with its corresponding list of `types`. At the end, the dictionary is then updated with this information and returned.

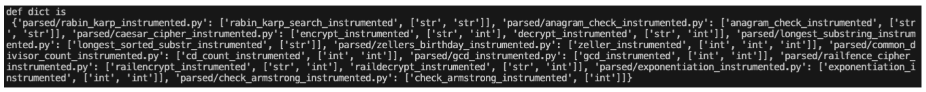
After the described function, the dictionary is further modified by removing from each signature unnecessary information such as the type the function returns and the parentheses with their values.



```
my dict is
 {0: ['rabin_karp_search_instrumented', ['str', 'str']], 1: ['anagram_check_instrumented', ['str', 'str']], 2: ['encrypt_instrumented', ['str', 'int'], 'decrypt_instru
mented', ['str', 'int']], 3: ['longest_sorted_substr_instrumented', ['str']], 4: ['zeller_instrumented', ['int', 'int', 'int']], 5: ['cd_count_instrumented', ['int', '
int']], 6: ['gcd_instrumented', ['int', 'int']], 7: ['railencrypt_instrumented', ['str', 'int'], 'raildecrypt_instrumented', ['str', 'int']], 8: ['exponentiation_instr
umented', ['int', 'int']], 9: ['check_armstrong_instrumented', ['int']]}
```

Figure 3: Get Type of Parameters Function result

Moreover, I have decided to create a new dictionary that maps each parsed Python file to its corresponding function signatures and parameter types for test case purpose.



```
def dict is
 {'parsed/rabin_karp_instrumented.py': ['rabin_karp_search_instrumented', ['str', 'str']], 'parsed/anagram_check_instrumented.py': ['anagram_check_instrumented', ['str
', 'str']], 'parsed/caesar_cipher_instrumented.py': ['encrypt_instrumented', ['str', 'int'], 'decrypt_instrumented', ['str', 'int']], 'parsed/longest_substring_instrum
ented.py': ['longest_sorted_substr_instrumented', ['str']], 'parsed/zellers_birthday_instrumented.py': ['zeller_instrumented', ['int', 'int', 'int']], 'parsed/common_d
ivisor_count_instrumented.py': ['cd_count_instrumented', ['int', 'int']], 'parsed/gcd_instrumented.py': ['gcd_instrumented', ['int', 'int']], 'parsed/railfence_cipher_
instrumented.py': ['railencrypt_instrumented', ['str', 'int'], 'raildecrypt_instrumented', ['str', 'int']], 'parsed/exponentiation_instrumented.py': ['exponentiation_i
nstrumented', ['int', 'int']], 'parsed/check_armstrong_instrumented.py': ['check_armstrong_instrumented', ['int']]}
```

Figure 4

Fuzzy Testing Function
This function intends to conduct fuzzy testing on Python functions. To do that, it explores different arguments for each function and stores the most effective ones. The function takes two dictionaries as input: the first contains for each parsed file the corresponding function signatures and parameter types, while the second one holds for each parsed file the corresponding code implementations.

The function iterates through each parsed file, extracting the function names and their variable types with which the script creates a pool of arguments. Then, the function executes the Python code with this pool and calculates distances to evaluate how many branches are covered.

Note that I have created the pool using the hyper-parameters provided in the lecture:

- `MIN_INT = -1000`
- `MAX_INT = 1000`
- `MAX_STRING_LENGTH = 10`
- `POOL_SIZE = 1000`

The script maintains archives for both true and false conditions, capturing the best arguments and distances. The pool is updated dynamically by adding a new argument as mutation or crossover (they have each 1/3 probability to be chosen) of the current one. The average pool size is recorded as a metric to maintain coherence with the hyper-parameters of the Genetic Algorithm.

A true list and a false list are then populated including the parsed Python file, the function with its corresponding best arguments, and results. These data are processed to facilitate the writing of test cases (both for true and false conditions).

Write Test Function
This function aims to write test cases based on a provided dictionary, a specified value, and a target folder. It iterates through file names and their associated function archives, creating import statements, class definitions, and individual test functions. It handles argument formatting and incorporates assert statements to verify the accuracy of tested functions. The test code generated is appended or written to the respective test files in the specified folder.

Lastly, to maintain the record of the Fuzzer execution, the `Fuzzy` folder, containing the test cases, is copied and pasted into an `Archive` folder. This is due to the statistical comparison of test generators in section 4. This process is performed for a specified number of copies that I have defined as hyper-parameter (`COPIES = 10`).

Section 3: Genetic Algorithm test generator

This section aims to describe the steps to generate the test cases using Genetic Algorithm (GA). The code section follows the next logic.

Main Loop
This code segment executes different processes to run a GA on the instrumented Python files, generating test cases and archiving the GA results.

Within each iteration, the code extracts

- the instrumented file;
- the corresponding code;
- both the corresponding function names and the type of the parameters.

It uses this information to compile and run the code.

Therefore, the script initializes archives for the coverage and both true and false branches. The GA is then executed for each function of the file, generating the archives (you can find the description of the execution in the paragraph GA Deap Function). The true and false branches are saved in separate lists, namely `list_true_archive` and `list_false_archive` where each element contains the file name, the current function, and the corresponding true or false branches.

These separated lists are then transformed into temporary dictionaries to simplify the test case generation process. This decision is driven by the desire to avoid a new dedicated function for writing Deap test cases; indeed, I use the same method already described for Fuzzy (paragraph: Write Test Function).

Lastly, to maintain the record of the GA execution, the `Deap` folder, containing the test cases, is copied and pasted into an `Archive` folder. This is due to the statistical comparison of test generators in section 4. This process is performed for a specified number of copies that I have defined as hyper-parameters (`COPIES = 10`).

GA Deap Function
This function is the central component of the GA. It uses global variables for critical information about true and false branches, the current file and the corresponding function, and lists to capture true and false branches for the writing test cases.

It initializes the Deap toolbox to surround the different operators and settings. The toolbox defines an individual using the `create_individual` function. Then, it initializes the population as a list of individuals. Moreover, using the `get_fitness` function, it evaluates the quality of the potential solutions. The toolbox also includes important operators such as crossover (`cross` function), mutation (`mut` function), and selection.

The genetic algorithm is executed for a specified number of repetitions (that is defined as hyper-parameter `REPS = 1`). For each run, the archive dictionaries for true and false branches are reset, and a population of individuals is initialized, while the Deap evolutionary algorithm is used to evolve the population.

Create Individual Function
This function aims to generate individuals. The function accesses the global variable `current_arg` which contains information about the expected variable types. It then iterates over each element of `current_arg`, representing the variable types in the function's signature. This allows to create random variables of the corresponding type by calling specific functions for each variable type, namely `init_int_variable` and `init_str_variable`. These variables are then added to a list that is returned as a tuple.

Get Fitness Function
This function aims to evaluate the fitness of an individual taken as input. The function accesses global variables such as the dictionaries for true and false branches, the current function, the total number of branches, and the archives for both true and false. To reset any values from previous executions, the distance dictionaries are cleared.

The function attempts to run the instrumented version of the current function under test with the provided arguments. Any special characters in the resulting string are replaced. If an AssertionError occurs during the function execution, the method returns a fitness value of infinity.

The initial fitness value is set to 0.0 increasing by the normalized distances for both the true and false branches. Lastly, the fitness value is returned.

Mutation Function
This function aims to facilitate the mutation operation. It takes an individual, `elem`, and delegates the mutation operation to a separate function, which receives this variable as input. The separate function, used also for Fuzzer, results in a modified version of the input assigned then to `elem`. The modified individual is lastly returned.

Crossover Function
This function performs the crossover operation by taking two individuals, `elem1` and `elem2` as input. The crossover logic is delegated to a separate function, which receives these variables as parents. The separate function, used also for Fuzzer, produces two offspring resulting in children assigned then to `elem1` and

`elem2`. The modified individuals are lastly returned.

Note that I have used the hyper-parameters provided in the lecture. The values chosen for `NPOP` and `NGEN` are due to maintain coherence with the average pool size of Fuzzer.

- `NPOP = 300`
- `NGEN = 10`
- `INDMUPROB = 0.05`
- `MUPROB = 0.3`
- `CXPROB = 0.3`
- `TOURNSIZE = 3`
- `LOW = -1000`
- `UP = 1000`
- `REPS = 1`
- `MAX_STRING_LENGTH = 10`

Section 4: Statistical comparison of test generators

This section aims to describe the results of the experimental procedure. Firstly, I collected 10 test folders each for Fuzzer and Deap, as mentioned earlier. This step is crucial for executing the mutation process using MutPy. Since I used a Macbook Pro with an ARM architecture (specifically, the M1 pro processor), I used the provided Google Colab to carry out this final part of the project.

The idea of the provided notebook is to clone the repository of Github on Colab computing the mutation score for each method (both Fuzzer and Deap) within every Python file. The obtained mutation scores are then saved in a text file named `mutation_scores.txt` which is downloaded and imported into my local project.

The main logic in my local project is the following. For each file, the script constructs a pandas DataFrame by combining the scores for Fuzzer and Deap into a unified list. The data frame contains columns for `File_Name`, `Score`, and `Metric`. The last column indicates whether the score is associated with Fuzzer or Deap. Then, the function calculates both Cohen's d-effect size and the Wilcoxon signed-rank test. Their outcomes are then used in a dedicated function that generates boxplots for the visual representation.

The comparison of mutation scores shows that both testing methods perform similarly. Fuzzer has a mean mutation score of 71.6, while GA achieves a slightly higher mean of 72.1. Nonetheless the results are not very high, the obtained values suggest robust test functions. Further improvements could be achieved by manual refinement and fine-tuning of test cases addressing limitations and enhancing the specificity of the tests.

- `anagram_check` $\rightarrow$ Fuzzer and Deap achieves the same mutation score average resulting in no significant difference.
- `caesar_chiper` $\rightarrow$ Despite the averages are different, the effect size and the p-value report a no significant difference.
- `check_armstrong` $\rightarrow$ Fuzzer and Deap mutation score averages are very close resulting in no significant difference as supporting by the effect size and the p-value.
- `common_divisor_count` $\rightarrow$ Fuzzer and Deap mutation score averages are very close resulting in no significant difference as supporting by the effect size and the p-value.
- `exponentiation` $\rightarrow$ Fuzzer and Deap achieves the same mutation score average resulting in no significant difference.
- `gcd` $\rightarrow$ Fuzzer and Deap mutation score averages are close resulting in no significant difference as supporting by the effect size and the p-value.

- `longest_substring` → Fuzzer and Deap mutation score averages are quite different resulting in a significant difference as supporting by the effect size and the p-value.
- `rabin_karp` → Fuzzer and Deap mutation score averages are very close resulting in no significant difference as supporting by the effect size and the p-value.
- `railfence_cipher` → Fuzzer and Deap mutation score averages are very different as supporting by the effect size. The Wilcoxon test results in no significant difference but note that the p-value is only 0.1.
- `zellers_birthday` → Fuzzer and Deap achieves the same mutation score average resulting in no significant difference.

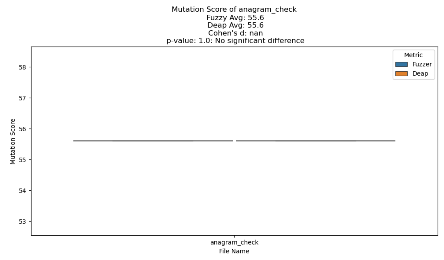


Figure 5: Steps for the last week - Light mode

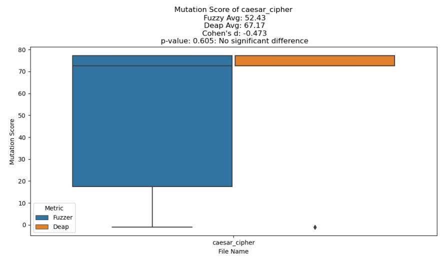


Figure 6: Steps for the last week - Dark mode

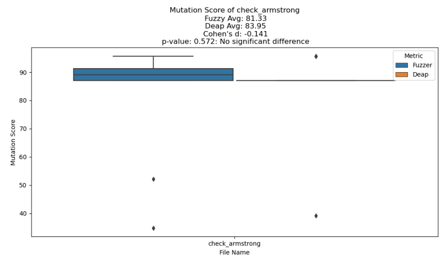


Figure 7: Steps for the last week - Light mode

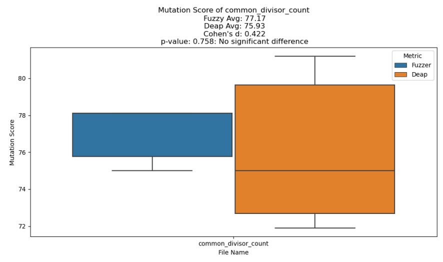


Figure 8: Steps for the last week - Dark mode

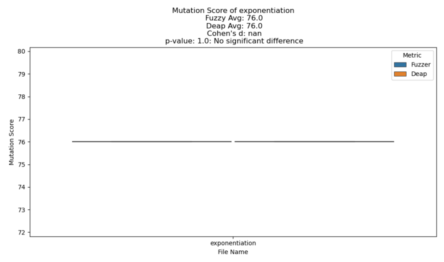


Figure 9: Steps for the last week - Light mode

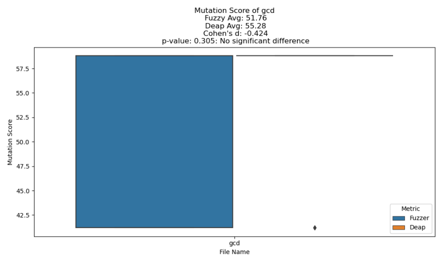


Figure 10: Steps for the last week - Dark mode

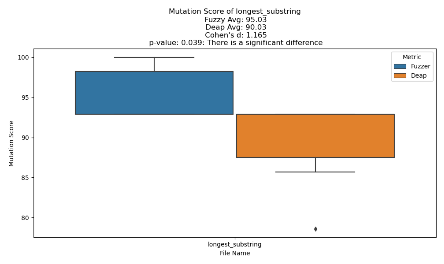


Figure 11: Steps for the last week - Light mode

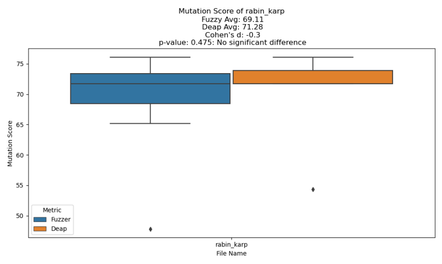


Figure 12: Steps for the last week - Dark mode

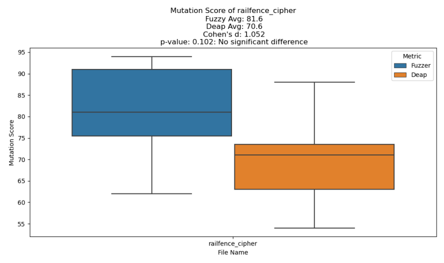


Figure 13: Steps for the last week - Light mode

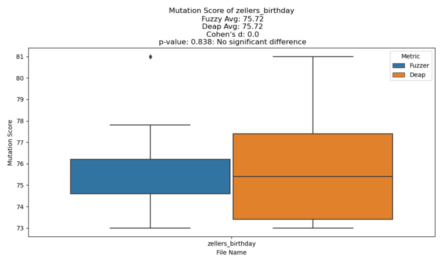


Figure 14: Steps for the last week - Dark mode

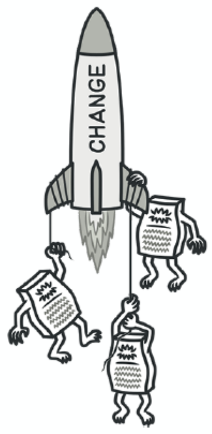


Figure 15: Caption.