Smart Contract Test Case Prioritization based on Frequency and Gas Consumption

Sangharatna Godboley*, P. Radha Krishna, Aditya Joshi, Ishita Gupta, Rahul Khatav
NITMiner Technologies, Department of Computer Science and Engineering
National Institute of Technology Warangal, Warangal, India
{sanghu,prkrishna}@nitw.ac.in, {joshi_851975,gupta_911910,khatav_861974}@student.nitw.ac.in
*corresponding author

Abstract—Testing is an integral process in the development life cycle of a Smart Contract, especially considering the immutable nature of blockchains. Thus, rigorous testing of smart contracts is necessary to identify defects or vulnerabilities and correct them before deployment. This proactive approach prevents any unfair advantages that may be exploited by one or more entities within the smart contract. In this paper, we present a three-layered approach for prioritizing test cases using gas consumption values and frequency of test case targeting methods, aiming to enhance the efficiency of the testing process. We illustrate our approach by applying to the smart contract *Ballot.sol*. We used the Ethereum Virtual Machine environment, and generated test cases using the Bounded Model Checker engine of the Solidity compiler. Additionally, we created manual test cases to simulate real-time smart contract behavior. We use the test cases of the *Ballot.sol* contract to showcase the prioritized list of test cases along with their respective individual and cumulative statement coverages. Our approach offers a faster testing environment for smart contract deployment.

Keywords–Smart Contracts; Test Case Suite; Gas; Frequency

1. Introduction

Smart contracts play an important role in blockchain technology. It enables self-executing agreements with predefined rules and conditions. As smart contracts continues to grow across various industries, ensuring their reliability and security becomes difficult. By comprehensive testing this critical aspect can be assured.

Testing smart contracts involves the creation of test cases that assess their functionality, efficiency, and security. However, not all test cases are created equal. Some may trigger more complex code paths or consume more gas than others, impacting the efficiency and effectiveness of the testing process [2].

In response to this challenge, this paper introduces an approach for Smart Contract Test Case Prioritization based on Frequency and Gas Consumption. Our method aims to optimize the testing process by selecting a subset of test cases that strike a balance between runtime efficiency and coverage. By considering the frequency of test case targeting and the gas consumption associated with each test case, we can prioritize the most critical test cases while reducing unnecessary redundancy.

We delve into the details of our approach, illustrating its application with the *Ballot.sol* smart contract. We explain how our algorithm selects the best subset of test cases to run, minimizing runtime without significantly compromising branch and statement coverage.

Checking for the correct execution of code requires test cases to be written for the smart contract. These smart contract test cases are written by testers and target particular parts of the code and match the output of the block with the expected output using assertion statements. These test cases might hold different amounts of significance to the process of testing, i.e. a particular test case might cause the execution of a larger segment of code (more functions) for a given input than some other test case. Further, a test case might also check the execution of multiple functions of the smart contract.

In addition to this, some functions might have to be executed multiple times to check output for different inputs. This suggests that different test cases might have different complexities in terms of gas consumption and method calls made. So, it is clear that different test cases have different priorities. It might hence not be possible to run all these test cases over and over again in the process of development. This gives us the motivation to devise an algorithm to find the best subset of test cases to run so that it reduces the run time and also does not compromise the branch/line coverage to a large extent.

The rest of the paper is organised as follows: Section 2 presents the basic concepts of the work and Section 3 discusses the relevant related work. Section 4 presents the proposed approach and Section 5 discusses the results analysis. Finally, Section 6 concludes the paper.

2. Background

In this section, we discuss fundamental concepts necessary to understand the proposed approach.

2.1. Ethereum Virtual Machine (EVM)

The Ethereum Virtual Machine (EVM) is a simulated blockchain environment that runs locally on a system, replicating the behavior of a blockchain. This platform proves exceptionally valuable for the development and testing of smart contracts. It consolidates all blockchain operations into

2693-9371/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS-C60940.2023.00084

one system, significantly accelerating transaction processing and reducing development and testing time. In this paper, all experiments have been conducted within the Hardhat environment [5].

2.2. Smart Contracts and Gas Consumption

Smart Contracts are essentially agreements deployed on the blockchain, each containing a set of conditions. Once these conditions are met, the predefined transaction is executed automatically. It is important to note that deploying these smart contracts on the blockchain requires altering the blockchain's state. Miners play a crucial role in this process, competing to win the bid for mining smart contracts by solving complex mathematical puzzles. The winning miner earns the right to execute the transaction. Solving these intricate puzzles consumes a significant amount of electricity. The complexity of a smart contract directly correlates with the cost of mining it. *Gas*, a computational unit, quantifies the state change in the blockchain.

2.3. Unit Test Cases for Smart Contracts

Unit test cases are code segments designed to test one or more parts of the code. This is done by providing inputs and verifying the outputs of smart contract methods against expected results, typically using assert statements as shown in Listing 1.

```
it("test case name or number", async function(){
    let result = SmartContractFactory.method(Input
    Parameters);
    assert.equal(result, expected output);
}
```

Listing 1: Example Solidity Test Case

For the purpose of this work, we have considered test cases generated by the Bounded Model Checker (BMC) engine of the Solidity compiler. These test cases are produced as counterexamples, which are then translated into test cases. Additionally, we have incorporated manual test cases to align with industry practices for test case development. This combination of automated and manual test cases provides a robust set of test cases for evaluating the algorithm.

2.4. Bounded Model Checker (BMC)

While the test cases can also be generated through alternative techniques or manual authorship, the proposed algorithm remains consistent across all cases. In the example test suite presented in this paper, test cases are generated using Bounded Model Checker (BMC), as well as manually [6].

3. Related Work

Regression testing is the process of assessing the functionality of code after the addition of new code or features. This practice is standard in the field of software testing and is critically important for the overall proper functioning of the code. Improvements to regression testing, such as the introduction of selection algorithms for identifying the best test case suite and selecting test cases based on the time complexity of algorithms for that test, have been previously developed in classical centralized system development environments [1]. These advancements can be categorized into two types of modifications, including algorithms like the execution slice technique [3] and modification-traversing tests [4].

Another category of regression testing is minimization or prioritization-based regression testing. This approach involves considering the most critical test cases. Various methods have been proposed to achieve this objective, ranging from slicing the most vital segments of the test cases to obtain a set of the most significant test cases [3] to prioritizing test cases based on their running time or time complexity. In the field of testing smart contracts, there are tools available, such as SolAnalyser[1], for identifying errors in smart contracts through code analysis and vulnerability detection. They also monitor gas consumption to determine when the gas limit has been reached, necessitating a rollback of the smart contract [7].

Regarding test case prioritization in classical systems, there are proposed approaches that utilize Software Agents and fuzzy logic. This approach aims to extract information from test cases to understand faults in the code and then prioritize them using this information [8].

Smith et al. [10] conducted a comparative analysis of test case prioritization techniques in the context of smart contract development. Brown et al. [11] focus on efficient test case prioritization specifically tailored for Ethereum smart contracts. It delves into techniques and strategies to prioritize test cases effectively, considering the unique characteristics of smart contract development.

Wang et al. [12] showed enhancements in smart contract testing by incorporating gas consumption analysis. Authors addressed on smart contract testing, that is, gas consumption analysis. This is to optimize the testing process by integrating an analysis of gas consumption which is an important factor in the execution of smart contracts on blockchain platforms. Due to the impact of gas consumption on the stability and performance of contracts, the authors propose new testing methodologies. This is important because it identifies and rectifies potential vulnerabilities.

Chen et al. [13] proposed a novel test case prioritization method for smart contracts based on static analysis techniques. Garcia et al. [14] provided strategies for Ethereum's contract topology. These will be helpful in testing strategies and priorities in the context of smart contracts.

Test case prioritization methods have been evolved for smart contracts to improve testing efficiency [15, 16]. Ma et al. [16] explored prioritization strategies specific to Ethereum smart contracts. Yang et al. [15] introduced a prioritization approach that considers code coverage and fault detection effectiveness.

Blockchain-based testing methods have emerged to address the unique challenges of smart contract testing [17, 18]. Tikhomirov et al. [17] introduced SmartSeeds, a tool for generating effective test cases, and Cheng et al. [18] presented *Chainr*, a framework for blockchain testing, including priori-

[1] https://github.com/sefaakca/SolAnalyser

tization features.

Empirical studies have been conducted to evaluate the effectiveness of test case prioritization techniques in the context of smart contracts [19, 20]. Amrhein et al. [19] conducted an empirical analysis of test case prioritization in Ethereum contracts, while He et al. [20] proposed a test prioritization approach based on historical test execution data.

Machine learning techniques have been applied to smart contract test case prioritization [21, 22]. Kholidy et al. [21] introduced an approach that utilizes machine learning models to predict test case execution times and prioritize accordingly, and Luo et al. [22] explored a similar concept for Ethereum contracts. The integration of test case prioritization into continuous integration and continuous deployment (CI/CD) pipelines has been investigated to automate testing workflows for smart contracts [23, 24]. Abbas et al. [23] discussed the incorporation of prioritization techniques into CI/CD processes, and Luo et al. [24] presented a CI/CD framework tailored for Ethereum smart contracts.

Godboley et al. [25] proposed an approach towards agile mutation testing using a branch coverage-based prioritization technique. They applied an efficient method to prioritize mutation testing for programs. This significantly reduces the testing time and resources which is needed for mutation analysis [25]. Monika et al. [28] developed a tool gMutant for faster mutation testing. Agarwal et al. [26] proposed a technique for cyclomatic complexity analysis of smart contracts using control flow graphs (CFG). They evaluate the complexity of smart contracts, which is crucial for identifying potential vulnerabilities and improving contract design [26]. Godboley [27] proposed SmartMuVerf which addresses the challenge of ensuring the correctness and security of smart contracts by verifying the behavior of mutants, providing a valuable tool for smart contract developers and auditors [27].

4. Proposed Approach

In this section, we present our proposed approach.

4.1. Motivation

This paper introduces an algorithm for prioritizing test cases based on gas consumption and frequency. Firstly, when a particular test case consumes more gas, it implies that more computation is necessary to change the state of the blockchain. This, in turn, suggests that the test case targets a larger portion of code or involves more complex code segments. Consequently, the specific test case or method under consideration becomes essential, as it is expected to cover more lines of code and deal with a greater number of code branches. This results in greater testing algorithm effectiveness due to increased line and branch coverage.

Secondly, if a particular part (method or data structure) appears in multiple test cases, it implies that this section of code holds particular interest to the tester (if manual test cases are written) or the model checker engine. Multiple checks with different input parameters are required, indicating that the methods or data structures targeted by more test cases are of greater importance.

Algorithm 1: Layer 1 of Prioritization

Input: fitnessValueOfMethod = [],
Layer1PrioritizedArray = [],minimumGas = ∞,
maximumGas = 0;

foreach *method* $\in$ *methodToGasConsumed* **do**
  if *method*$->$*gas* $>$ *maximumGas* **then**
    maximumGas $\leftarrow$ method$->$gas;
  if *method*$->$*gas* $<$ *minimumGas* **then**
    minimumGas $\leftarrow$ method$->$gas;

Set maximumTestCases = ∞, minimumTestCases = 0;

foreach *method* $\in$ *methodToTestCases* **do**
  if *method*$->$*testCasesNumber* $>$ *maximumTestCases* **then**
    maximumTestCases $\leftarrow$ method$->$testCasesNumber;
  if *method*$->$*testCasesNumber* $<$ *minimumTestCases* **then**
    minimumTestCases $\leftarrow$ method$->$testCasesNumber;

Set maximumTestCases = ∞, minimumTestCases = 0;

foreach *method* $\in$ *methodToTestCases* **do**
  if *method*$->$*testCasesNumber* $>$ *maximumTestCases* **then**
    maximumTestCases $\leftarrow$ method$->$testCasesNumber;
  if *method*$->$*testCasesNumber* $<$ *minimumTestCases* **then**
    minimumTestCases $\leftarrow$ method$->$testCasesNumber;

Set normalizedTestCase = $\emptyset$, normalizedGas = $\emptyset$,
newMax = 1, newMin = 1;

foreach *method* $\in$ *methods* **do**
  $frequencyNormalization \leftarrow ((method.testCasesNumber - minimumTestCases)/(maximumTestCases - minimumTestCases)) * (newMax - newMin) + newMin$;
  $gasNormalization \leftarrow ((method.gas - minumumGas)/(maximumGas - minimunGas)) * (newMax - newMin) + newMin$;
  $normalizedTestCase[method] \leftarrow frequencyNormalization$;
  $normalizedGas[method] \leftarrow gasNormalization$;

foreach *method* $\in$ *methodToTestCases* **do**
  $Layer1PrioritizedArray.add((method, methodTo-TestCases[method])$;

foreach *method* $\in$ *methodToTestCases* **do**
  $Layer1PrioritizedArray.sort(key = lambdax : 0.5 * normalizedTestCases[x[0]] + 0.5 * normalizedGas[x[0]], reverse = True)$;

resultantBoxes = $\emptyset$;

foreach *box* $\in$ *Layer1PrioritizedArray* **do**
  $resultantBoxes.add(box[1])$;

return $resultantBoxes$;

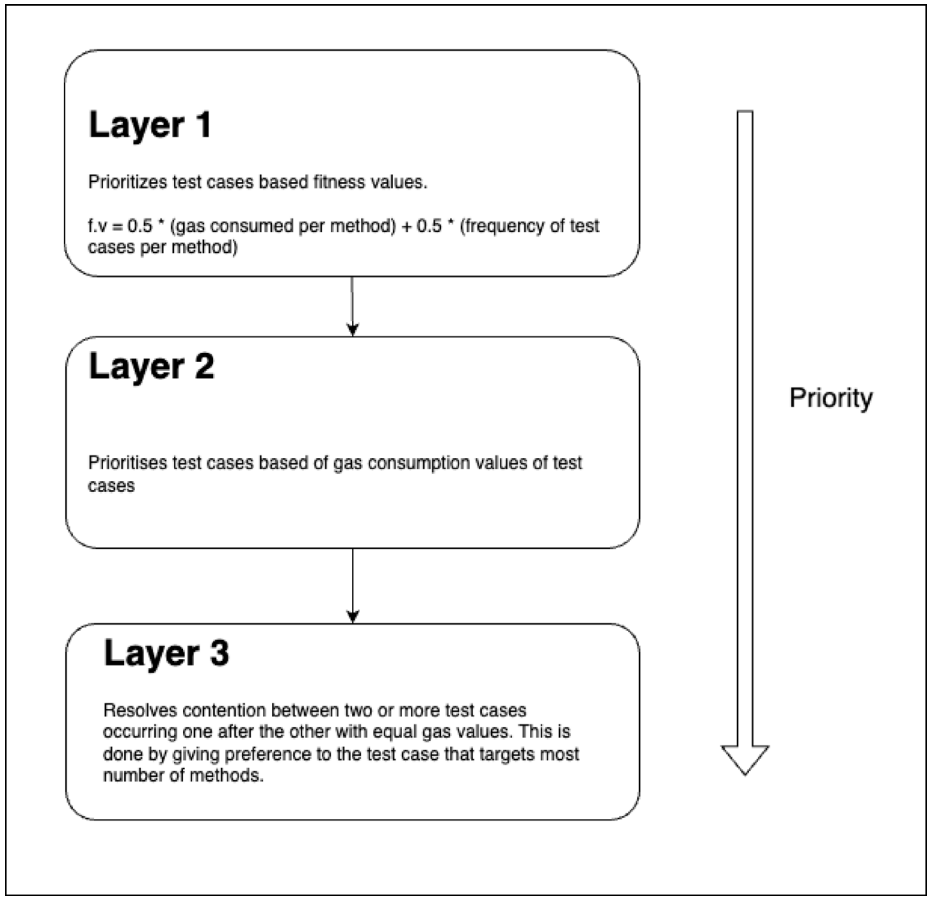

Figure 1: Framework of Propsoed Idea

4.2. Prioritization Algorithm

The proposed idea follows a three-layered approach. It operates sequentially, which means it prioritizes Layer 1 first, followed by Layer 2, and lastly Layer 3 as shown in Fig. 1.

4.3. Layer 1

Algorithm 1 illustrates the implementation of Layer 1. In Layer 1 of the idea, we consider the average gas consumed by each method and the number of test cases that invoke each particular method. Since gas values are often very large, and the number of test cases calling a method varies, we normalize these values to fall within the range of [0, 1]. We iterate through the methods to calculate these values for all functions, storing them in two separate data structures.
Once these values are normalized, a fitness function denoted by **f.v** is defined in Eq. 1.

$$\boxed{f.v = (methodGas_i) \times 0.5 + (testCasesPerMethod_i) \times 0.5} \quad (1)$$

Here, $methodGas_i$ represents the gas consumed by the i-th method, and $testCasesPerMethod_i$ represents the number of test cases targeting that method, which is also referred to as the frequency value. $methodGas_i$ is constrained to the range [0, 1].

Next, we create another data structure that stores the test cases for each of the methods. Let us assume that these are the boxes, where each box contains the number of test cases for a method j, where j belongs to N (the total number of methods in the Smart Contract). These boxes are indexed by their respective methods and then sorted in descending order based on the value of each method's fitness function. The result of Layer 1 is a two-dimensional array with test cases in each row.

It is important to note that in Layer 1 of the algorithm, the weight values (coefficients for gas value and frequency value) are both set to *0.5*. We have selected these weights for our algorithm to equally prioritize both gas consumption and frequency in the first layer of assessment. It's worth recognizing that there are no universally applicable weight values for prioritizing test cases since the prioritization may need to be tailored to the specific requirements of the smart contract under consideration. Therefore, these weight values are adjustable. However, it is crucial to maintain the constraint that the sum of these two weights equals 1. This constraint is necessary to ensure that the values stay within a normalized range of 0 to 1, achieved through min-max normalization, as applied to both gas values and frequency values. For these purposes, we used weight values of *0.5* for both gas and frequency in this paper.

Algorithm 2: Layer 2 of Prioritization

Input: resultantPrioritizedTestCaseListAfterLayer2 = $\emptyset$;
foreach *method* $\in$ *TCRankingAfterFilteringBasedOnTestCasesPerMethod* **do**
    $method.sort(key = lambda x : x.gasConsumption, reverse = True)$;
    foreach *TestCase* $\in$ *method* **do**
        if *TestCase* $\notin$ $resultantPrioritizedTestCaseListAfterLayer2$;
        then
            $resultantPrioritizedTestCaseListAfterLayer2.add(TestCase)$;
    end
end
return resultantPrioritizedTestCaseListAfterLayer2

4.4. Layer 2

Algorithm 2 illustrates the implementation of Layer 2. After receiving the 2-dimensional array from the first layer, the initial task is to eliminate redundant test cases. Let $PrioritizedTestCasesAfterLevel1$ represent the 2-dimensional array obtained from Layer 1. If there are i rows and j columns, then the number of columns in $PrioritizedTestCasesAfterLevel1$ is j. If a test case is redundant in two rows, let's say in rows k and l where k is less than l, we remove the redundant test case from row l. This ensures that the Layer 2 approach does not override the priority of Layer 1. Next, we concatenate all the rows of $PrioritizedTestCasesAfterLevel1$ into a 1-dimensional array. Then, we sort this 1-D array in descending order based on the gas consumed by each particular test case. Let's call this 1-dimensional array $GasPrioritizedTestCases$.

4.5. Layer 3

Algorithm 3 demonstrates the implementation of Layer 3. After Level 2, it is possible that two test cases with the same priority are placed consecutively, following a first-come, first-served basis. However, this is not desirable. So, we introduce another parameter: $numberOfMethodsTestCaseHasTargeted$. We employ the two-pointer approach to address this issue. If there exists a sub-array in the $GasPrioritizedTestCases$ array where all the test cases have the same gas consumption, we sort these test cases in descending order based on their gas values within that specific sub-array. This layer eliminates redundancies of any type to produce a prioritized list of test cases.

5. Experimental Study

In this section, we discuss a case study with the Usage, Walkthrough, and results.

Algorithm 3: Layer 3 of Prioritization

Input: i $\leftarrow$ 0, j $\leftarrow$ 1 ;
while $i <$ *TCRankingAfterApplyingGasConsumptionPrioritization.size() and* $j <$ *TCRankingAfterApplyingGasConsumptionPrioritization.size()* **do**
    hasIncreased $\leftarrow$ 0 ;
    while *testCaseToGas[TCRankingAfterApplyingGasConsumption Prioritization[j]] == testCaseToGas[TCRankingAfterApplyingGasConsumption Prioritization[i]] and j < SIZEOF(TCRankingAfterApplyingGas ConsumptionPrioritization)* **do**
        $j \leftarrow j + 1$;
        hasIncreased $\leftarrow$ 1 ;
    end
    if *hasIncreased == 1* **then**
        TCRankingAfterApplyingGasConsumption Prioritization[i:j].sort(key:lambda, numberOfMethodsTheParticularTestCaseIsTargetting,reverse = True);
    end
end
return TCRankingAfterApplyingGasConsumptionPrioritization

TABLE I: Methods in Ballot smart contract

#M	MName	AGas	f.v
1	giveRightToVote	340551	0.215716
2	delegate	789349	0.5
3	vote	165918	0.105098
4	winingProposal	0	0
5	winnerName	0	0
6	chairPerson	0	0
7	voters	2	0.500001
8	Proposals	0	0

5.1. Usage

We utilized the Hardhat EVM (Ethereum Virtual Machine)[2] for this paper. Our test cases were generated using the Bounded Model Checker of the Solidity Compiler, supplemented by a few custom test cases to simulate real-time system scenarios. The primary smart contract employed in this context is *Ballot.sol* [9].

The average gas consumption values for each method are shown in Table I. The *Ballot.sol* smart contract includes a total of eight methods. Column 1 of Table I shows the Method Number represented by *#M* and Column 2 shows the Name of the Method represented by *MName*. Column 3 shows the

[2]https://hardhat.org/hardhat-network/docs/overview

TABLE II: Original

TId	GAS
1	0
2	0
3	48645
4	48645
5	48645
6	48645
7	71759
8	71759
9	48657
10	48657
11	0
12	71759
13	71759
14	0
15	71759
16	71759
17	71759
18	71759
19	0
20	92915
21	48657
22	0
23	73003
24	71759
25	71759
26	0
27	71759

TABLE III: Prioritised

PTId	GAS
20	92915
27	71759
25	71759
24	71759
7	71759
8	71759
12	71759
13	71759
15	71759
16	71759
17	71759
18	71759
9	48657
10	48657
21	48657
4	48645
6	48645
5	48645
3	48645
14	0
11	0
22	0
26	0
23	73003
1	0
2	0
19	0

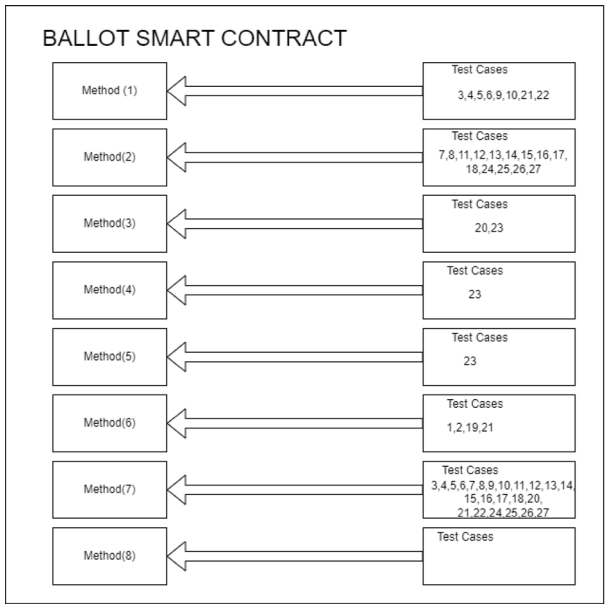

Figure 2: Methods and their associated test cases

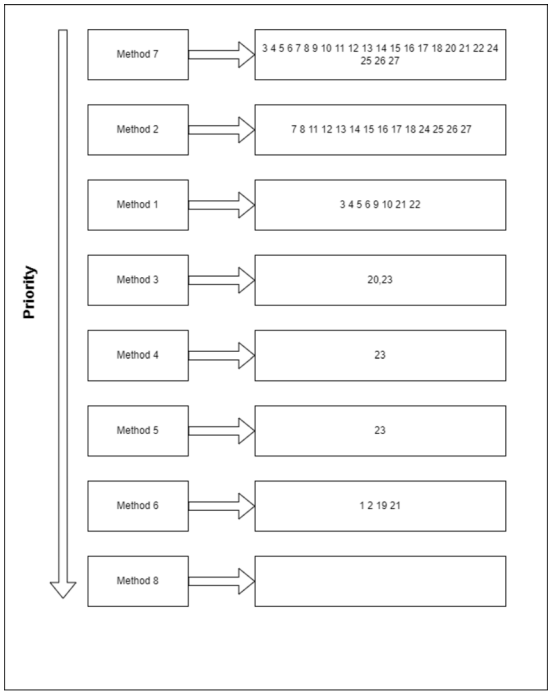

Figure 3: Output of Layer 1

average gas consumption represented by *AGas*. Column 4 shows *f.v* fitness values for all the methods using Eq. 1. To maintain simplicity, we use the serial numbers of the test cases to refer to the methods in the subsequent discussion.

The Computed Gas for the Test Cases are shown in Table II. The order of the test is considered Original, that is, as per the sequence of test cases. In Table II, Column 1 shows *TId* represents Test Case Id and Column 2 shows *GAS* which is the amount of GAS consumed for a test case.

The test cases that target specific methods in this case study are presented in Fig. 2. It is important to observe that some test cases target multiple methods. This occurs when a single test case is designed to assess the interaction between multiple methods.

5.2. Walk-Through

Now, consider the data outlined in the previous section and systematically walk through the algorithm.

In Layer 1, the algorithm prioritizes the methods based on their fitness values. To accomplish this, it calculates the fitness values for each of the methods and arranges them in sorted order. In our case study, the methods are sorted in descending order of their respective fitness values, as illustrated in Fig. 3.

Next, we progress to Layer 2. In this layer, the algorithm arranges each of the test cases that target specific methods by their gas consumption in descending order. This sorting process is depicted in Fig. 4. As a result, the test cases are now organized in a relatively more sorted and prioritized manner, as opposed to being entirely random.

In Layer 3 of prioritization, the algorithm focuses on identifying two or more adjacent test cases with the same gas

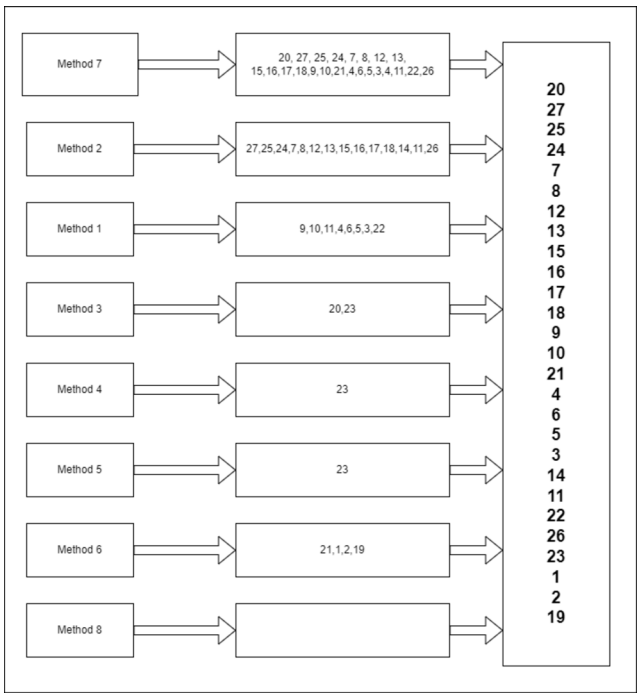

Figure 4: Output of Layer 2

consumption. It then prioritizes the test cases that target a higher number of methods. This prioritization approach is detailed in Table III. The order of test cases is as per the Prioritisation of Test Cases based on computed GAS. Column 1 shows *PTId* which represents Prioritised Test Case Id. Column 2 shows GAS which represents the amount of GAS consumed for a test case.

5.3. Results

Table IV is intended to provide an understanding of the cumulative code coverage of the prioritised test cases. We target to save time and resources during the testing process while not substantially reducing the effort required for regression testing. Columns in Table IV show the rankings of the test cases T^i, where i is the test case position. The upper triangle of the first cell in Column 1 shows the Rank, and the lower triangle of the first cell shows *#TCs* i.e. total number of collected test cases. Cells colored with *Magenta* show that the positions for the test cases are empty. It means, these test cases are not considered to compute the Cumulative Statement Coverage.

Consider that "*Y%*" represents the maximum code coverage achieved by all the test cases combined. Also, note that each individual test case has a certain code coverage, denoted as "X%" (the value of X may vary for each test case). The statement coverage of each individual test case as per Original Order is provided in Fig. 5. With the list of prioritized test cases, our goal is to demonstrate the effectiveness of our algorithm by showing that we can attain the maximum possible code coverage percentage, *Y%* through a set of test cases taken in the prioritized order. This subset of test cases is now smaller than what we initially had. This will lead to time and effort savings during regression testing since we would only use this particular subset of test cases that provides maximum code coverage.

It is essential to note that this approach is strictly reliant on the code coverage aspect, which is derived from the frequencies and gas consumption values of methods and test cases. This approach compromises if the tester has written test cases that focus on checking specific edge cases' functionality rather than covering the code.

There are a few interesting cases to discuss, which are highlighted in green color, in Table IV. The Test Case suite comprises *27* test cases.

The first scenario, considers only 1 test case which is the highest ranked test case i.e *TId 20* which leads to 27.78% of statement coverage. The detailed coverage report is shown in Listing 2. Almost, one-fourth of overall code coverage is taken with this test case alone. Assume that, the budget of test cases was approx. 28% code coverage then only this test case was sufficient, and the other 26 test cases would have been discarded.

```
percentage of statements covered: 27.78
percentage of branches covered: 10
percentage of functions covered: 33.33
percentage of lines covered: 31.25
```

Listing 2: Prioritization Coverage report for top 1 TC

Now, adding the next high ranked test case i.e. TId 27, the statement coverage achieved is 61.11%. The detailed coverage report is shown in Listing 3. Considering this scenario, only 2 test cases are power full enough among all other test cases to achieve more than 60% of overall code coverage.

```
percentage of statements covered: 61.11
percentage of branches covered: 25
percentage of functions covered: 50
percentage of lines covered: 59.38
```

Listing 3: Prioritization Coverage report for to 2 TCs

Further, if we keep on adding the test cases from TCs 3 to 12 then the cumulative statement coverage remains same i.e. 61.11%. The detailed coverage report is shown in Listing 4. But as soon as we add another test case i.e. 13th TId then the cumulative statement coverage achieved is 77.78%. After prioritization by our algorithm, considering only the top 13 test cases {20,27,25,24,7,8,12,13,15,16,17,18,9}, the coverage report is shown in Listing 3. Almost 50% of 27 test cases achieved approx. 78% of cumulative statement coverage, which is significant result.

Next, we can observe TId 14th to 23rd viz. {10, 21, 4, 6, 5, 3, 14, 11, 22, 26} are ineffective to achieve more statement coverage. But, as soon as we add TId 24th i.e. 23 then cumulative statement coverage achieved becomes 94.44% which is the maximum coverage achieved. The detailed coverage report is shown in Listing 5. Later, we can observe that, TIds 25th to 27th are ineffective. Also, the total coverage for all 27 test cases is shown in the Listing 5.

Importantly, by this analysis we can choose a set of test cases that are really contributing to maximal statement coverage as per the budget.

TABLE IV: Results

Rank / #TCs	T^1	T^2	T^3	T^4	T^5	T^6	T^7	T^8	T^9	T^{10}	T^{11}	T^{12}	T^{13}	T^{14}	T^{15}	T^{16}	T^{17}	T^{18}	T^{19}	T^{20}	T^{21}	T^{22}	T^{23}	T^{24}	T^{25}	T^{26}	T^{27}	Cov.
1	20																											27.78
2	20	27																										61.11
3	20	27	25																									61.11
4	20	27	25	24																								61.11
5	20	27	25	24	7																							61.11
6	20	27	25	24	7	8																						61.11
7	20	27	25	24	7	8	12																					61.11
8	20	27	25	24	7	8	12	13																				61.11
9	20	27	25	24	7	8	12	13	15																			61.11
10	20	27	25	24	7	8	12	13	15	16																		61.11
11	20	27	25	24	7	8	12	13	15	16	17																	61.11
12	20	27	25	24	7	8	12	13	15	16	17	18																61.11
13	20	27	25	24	7	8	12	13	15	16	17	18	9															77.78
14	20	27	25	24	7	8	12	13	15	16	17	18	9	10														77.78
15	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21													77.78
16	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21	4												77.78
17	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21	4	6											77.78
18	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21	4	6	5										77.78
19	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21	4	6	5	3									77.78
20	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21	4	6	5	3	14								77.78
21	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21	4	6	5	3	14	11							77.78
22	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21	4	6	5	3	14	11	22						77.78
23	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21	4	6	5	3	14	11	22	26					77.78
24	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21	4	6	5	3	14	11	22	26	23				94.44
25	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21	4	6	5	3	14	11	22	26	23	1			94.44
26	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21	4	6	5	3	14	11	22	26	23	1	2		94.44
27	20	27	25	24	7	8	12	13	15	16	17	18	9	10	21	4	6	5	3	14	11	22	26	23	1	2	19	94.44

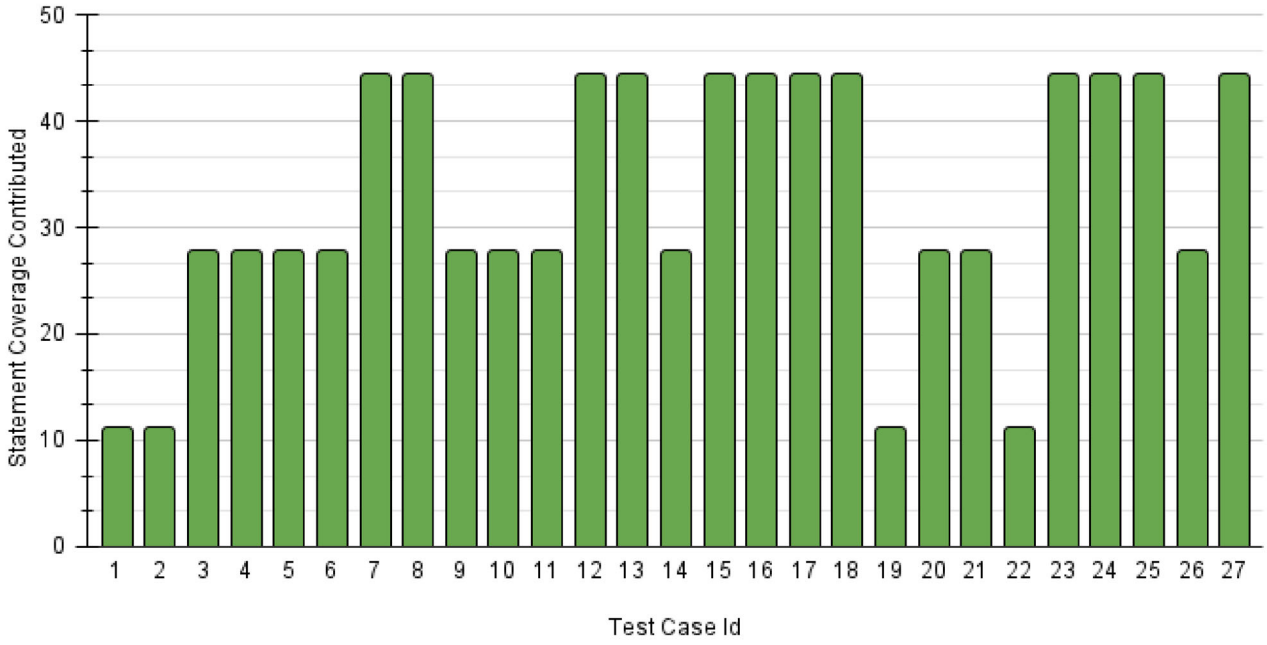

Figure 5: Statement Coverage Contributed

```
percentage of statements covered: 77.78
percentage of branches covered: 40
percentage of functions covered: 66.67
percentage of lines covered: 71.88
```

Listing 4: Prioritization Coverage report for top 13 TCs

```
percentage of statements covered: 94.44
percentage of branches covered: 55
percentage of functions covered: 100
percentage of lines covered: 90.63
```

Listing 5: Coverage report for top 23 TCs

6. Conclusion

The proposed three-layered prioritization algorithm for test cases in the context of smart contracts presents a systematic and data-driven method to optimize the testing process. We considered gas consumption and test case frequency for identifying and prioritizing test cases. This provides comprehensive code coverage. This approach not only enhances the efficiency of testing but also ensures that critical code segments are thoroughly examined. Prioritization of test cases was strategically demonstrated through various scenarios, after highlighting its effectiveness in achieving substantial code coverage with a limited set of tests. Utilising resources efficiently can be done by discarding redundant or less influential test cases. In conclusion, this prioritization algorithm stands as a valuable tool for the blockchain development community. This facilitates more effective and efficient testing practices and ultimately contributes to the overall reliability and security of smart contracts and blockchain-based applications. In the future, we extend this work with rigorous experiments with more smart contracts. We will conduct an analysis to show how the prioritised test cases are of quality. Also, we will work on a new fitness formula with more factors.

Acknowledgement

This work is sponsored by IBITF, Indian Institute of Technology (IIT) Bhilai, under the grant of PRAYAS scheme, DST, Government of India.

References

[1] Wong, W. Eric, Joseph R. Horgan, Saul London, and Hiralal Agrawal. "A study of effective regression testing in practice." In PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering, pp. 264-274. IEEE, 1997.

[2] Signer, Christopher. Gas cost analysis for ethereum smart contracts. MS thesis. ETH Zurich, Department of Computer Science, 2018.

[3] Agrawal, Hiralal, Joseph Robert Horgan, Edward W. Krauser, and Saul A. London. "Incremental regression testing." In 1993 Conference on Software Maintenance, pp. 348-357. IEEE, 1993.

[4] Rothermel, Gregg. Efficient, effective regression testing using safe test selection techniques. Clemson University, 1996.

[5] Hildenbrandt, Everett, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore et al. "Kevm: A complete formal semantics of the ethereum virtual machine." In 2018 IEEE 31st Computer Security Foundations Symposium (CSF), pp. 204-217. IEEE, 2018.

[6] Biere, Armin, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. "Bounded model checking." Handbook of satisfiability 185, no. 99 (2009): 457-481.

[7] Akca, Sefa, Ajitha Rajan, and Chao Peng. "SolAnalyser: A framework for analysing and testing smart contracts." 2019 26th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2019.

[8] Malz, Christoph, Nasser Jazdi, and Peter Gohner. "Prioritization of test cases using software agents and fuzzy logic." 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE, 2012.

[9] Ballot.sol Smart Contract https://docs.soliditylang.org/en/v0.8.11/solidity-by-example.html

[10] Smith, J., & Johnson, A. (2018). "A Comparative Analysis of Test Case Prioritization Techniques in Smart Contract Development." In Proceedings of the International Conference on Blockchain Technology (ICBT) (pp. 123-136). ACM.

[11] Brown, M., & Clark, L. (2020). "Efficient Test Case Prioritization for Ethereum Smart Contracts." Journal of Blockchain Research, 4(1), 25-37.

[12] Wang, Q., & Li, X. (2019). "Enhancing Smart Contract Testing with Gas Consumption Analysis." In Proceedings of the International Conference on Software Engineering (ICSE) (pp. 281-294). IEEE.

[13] Chen, Y., & Wu, Z. (2021). "A Novel Test Case Prioritization Approach for Smart Contracts Based on Static Analysis." Journal of Computer Science and Technology, 36(5), 1127-1144.

[14] Garcia, R., & White, J. (2017). "Analyzing Ethereum's Contract Topology." In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS) (pp. 1439-1453). ACM.

[15] Xiaohong Yang, Linzhang Tan, Qinghua Xie, and Xue Li. Test Case Prioritization for Smart Contracts. *Journal of Computer Science and Technology*, 34(4):796–813, 2019.

[16] Xiaoyue Ma, Jing Chen, and Zibin Zheng. Towards Automated Test Prioritization for Ethereum Smart Contracts. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 152–168, 2018.

[17] Sergei Tikhomirov, Maria Christakis, and Ranjit Jhala. SmartSeeds: Practical Seed Suggestion for Smart Contract Fuzzing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1483–1500, 2018.

[18] Lei Cheng, Xun Li, Yifan Zhou, Zijiang Liu, and Ping-Chen Hsiung. Chainr: A Framework for Testing Blockchain-based Systems. In *International Conference on Software Engineering*, pages 31–41, 2020.

[19] Sven Amrhein, Marcel Böhme, Lorenz Breidenbach, and Elisa Tischhauser. Empirical Evaluation of Test Case Prioritization for Ethereum Smart Contracts. *Journal of Systems and Software*, 157:110379, 2019.

[20] He, Z., Cai, Y., Wang, P., & Jin, Z. (2019). Towards improving software testing with historical failure information. In Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (pp. 813-823).

[21] Kholidy, H. S., Harman, M., & Islam, S. Z. (2019). Smart contract test case prioritisation. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) (pp. 189-198).

[22] Luo, C., Li, X., & Cheng, J. (2020). A novel method for improving smart contract testing using machine learning. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (pp. 191-201).

[23] Abbas, N., Chiba, D., & Shin, Y. (2019). Smart contract test generation and execution. In 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB) (pp. 7-12).

[24] Luo, C., Li, X., Cheng, J., Ma, T., Xie, T., & Zou, J. (2021). A systematic solution for testing and deploying smart contracts. ACM Transactions on Software Engineering and Methodology (TOSEM), 30(2), 1-32.

[25] Godboley, S., Mohapatra, D.P. (2022). Towards Agile Mutation Testing Using Branch Coverage Based Prioritization Technique. In: Przybyłek, A., Jarzębowicz, A., Luković, I., Ng, Y.Y. (eds) Lean and Agile Software Development. LASD 2022. Lecture Notes in Business Information Processing, vol 438. Springer, Cham. https://doi.org/10.1007/978-3-030-94238-0_9

[26] Agarwal, S., Godboley, S., Krishna, P.R. (2022). Cyclomatic Complexity Analysis for Smart Contract Using Control Flow Graph. In: Panda, S.K., Rout, R.R., Sadam, R.C., Rayanoothala, B.V.S., Li, KC., Buyya, R. (eds) Computing, Communication and Learning. CoCoLe 2022. Communications in Computer and Information Science, vol 1729. Springer, Cham. https://doi.org/10.1007/978-3-031-21750-0_6

[27] Godboley S. and Krishna P. (2023). SmartMuVerf: A Mutant Verifier for Smart Contracts. In Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE, ISBN 978-989-758-647-7, SciTePress, pages 346-353. DOI: 10.52200011822200003464

[28] Monika Rani Golla and Sangharatna Godboley. 2023. GMutant: A gCov based Mutation Testing Analyser. In Proceedings of the 16th Innovations in Software Engineering Conference (ISEC '23). Association for Computing Machinery, New York, NY, USA, Article 22, 1–5. https://doi.org/10.1145/3578527.3578546