Greenify: A Game with the Purpose of Test Data Generation for Unit Testing

Sharmin Moosavi, Hassan Haghighi, Hasti Sahabi, Farzam Vatanzade, and Mojtaba Vahidi Asl(✉)

Faculty of Computer Science and Engineering, Shahid Beheshti University, G. C., Evin, Tehran, Iran
mo_vahidi@sbu.ac.ir

Abstract. One of the most important, but tedious and costly tasks of software testing process is test data generation. Several methods for automating this task have been presented, yet due to their practical drawbacks, test data generation is still widely performed by humans in industry. In our previous work, we employed the notion of Game With A Purpose (GWAP) and introduced Rings as a GWAP to reduce time and costs of human-based test data generation and increase its appeal to engage even nontechnical people. In this paper, we propose a new game, called Greenify, with the purpose of test data generation so that it solves the main issues of Rings. The environment of this game is built based on a program's control flow graph. To evaluate the proposed approach, we designed several game levels based on six different C++ programs and gave them to volunteering players. The results show that in comparison to both conventional human-based approach and Rings, Greenify generates test data with less rime for all feasible paths of the given benchmark programs. In addition, Greenify identifies the smaller set of likely infeasible paths.

Keywords: Test data generation · Game With A Purpose · Human-based computation game · Human-based software testing

1 Introduction

Most software systems have numerous possible choices for test data when being tested. Thus, various criteria have been defined to find as small as possible subset of input values that result in more effective tests, i.e., those tests which lead to finding more failures of the Software Under Test (SUT). This process is called test data generation [1].

Three different categories of automatic test data generation methods have been presented including the random-based, symbolic execution, and search-based methods. These approaches have some drawbacks [3, 4, 12, 17] that makes them still incomplete and ineffective in producing test data [5], and therefore, test data generation is still typically carried out by human experts. Generating test data by humans has several advantages among which is the ability of humans to understand and interpret the code being tested while the computing power of human mind helps him solve problems with

H. Hojjat and M. Massink (Eds.): FSEN 2019, LNCS 11761, pp. 77–92, 2019.
https://doi.org/10.1007/978-3-030-31517-7_6

high levels of complexity. However, generating data for large SUT is very difficult for humans. In addition, test data generation by humans is a very tedious, time-consuming and costly task in the software development process [1, 2].

Nowadays, one of the well-known methods for solving problems is broadcasting them to the crowd. Some of the crowdsourcing models use computer games in which mind challenges are one of their most important elements. As player thinks and deducts in the process of a game, he can implicitly and through no knowledge of his own, help solve other problems (for example, test data generation) which do not have entertainment goals, by their own. In these situations, crowdsourcing and human-based computation help extracting a significant amount of information from a large number of players and users [7, 11, 14, 16]. This approach to solve problems introduces the concept of Game With A Purpose (GWAP). Based on this idea, the method suggested in this paper for test data generation involves designing a game and extracting test data based on solutions each non-technical player finds implicitly when playing the game.

The first and only attempt for employing GWAP for test data generation was through the "Rings" game introduced in [13]. The puzzles of Rings are designed based on symbolic execution technique. For each path constraint in a program unit's Control Flow Graph (CFG), a Rings's puzzle is generated. When a player solves the puzzle, he is implicitly generating appropriate input values that lead to the execution of the chosen CFG path.

Rings alleviates some problems of human-based test data generation. Since the shape of the game is not technical, bigger problems can be solved using crowdsourcing (by nontechnical people without getting paid) with a major reduction in costs of test data generation. Furthermore, the lack of motivation is compensated with the amusement of the game. The main drawback of Rings is incapability of visualizing programs with complex conditional statements. The other shortcoming is the disposal of wrong solutions although they could be right solutions of other paths.

In this paper, we aim at designing a GWAP based on concrete execution that does not suffer from the Ring's problems. In the proposed game, called Greenify, the environment is built based on CFG of a given program. The CFG's nodes are displayed by light bubbles, and a path of the CFG is represented by a string of connected light bubbles. The players should change the input power flows of the string until the color of all connected bubbles turns to green. During the gameplay, players are actually generating necessary data to cover the given test paths. Greenify can mitigate the main drawbacks of Rings since:

1- It is not based on symbolic execution.
2- It stores all acquired data of player quests, even if it does not lead to success.
3- By checking all data extracted from different players' play, the likely infeasible paths could be identified.

We have conducted an experiment to evaluate the Greenify performance in comparison to Rings, the conventional human-based and random approaches. We have selected six programs and developed the puzzles of both Greenify and Rings for all paths of these programs. Then, we asked a group of players to play the puzzles, and also, a group of programmers to manually generate data. The highlights of the experimental results are as follows:

- When players play Greenify, they generate data faster than other approaches.
- Many Greenify puzzles are solved by wrong data obtained for other puzzles.
- All feasible paths are covered by Greenify.
- The smaller set of probable infeasible paths is identified by Greenify.

In the following, we glance through related work in Sect. 2. The design of Greenify is presented in Sect. 3. In Sect. 4, we have provided the experimental results. Finally, Sect. 5 is devoted to the conclusions and some directions for future work.

2 Related Work

One of the methods for crowdsourcing is using game thinking. Hence, in this section, some important works that apply the concept of crowdsourcing and game thinking in the field of software testing are presented (Fig. 1).

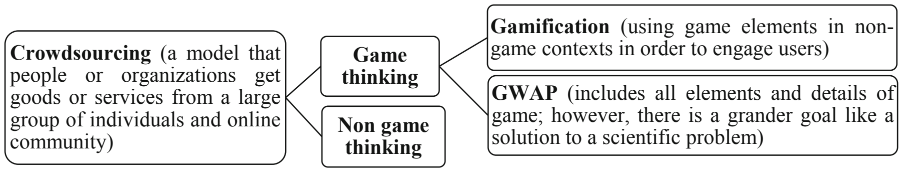

Crowdsourcing (a model that people or organizations get goods or services from a large group of individuals and online community)
Game thinking
Non game thinking
Gamification (using game elements in non-game contexts in order to engage users)
GWAP (includes all elements and details of game; however, there is a grander goal like a solution to a scientific problem)

Fig. 1. The methods for crowdsourcing

Crowdsourcing- Nowadays crowdsourcing is a popular method in the domain of software testing. In [26, 27], some complex testing tasks in functional testing and verification such as cross-browser verification have been given to the crowd. The results show that test performance is improved in terms of time and bug detection compared to traditional software testing [28]. The research in [29, 30] use crowd-sourced software testing (CST) for validation and acceptance testing and the results indicate that CST improves quality while being more flexible. CST is shown to be reliable, cost-efficient with high quality [30, 31] in usability testing, as well. To our knowledge, there is no CST for test data generation in the literature.

Gamification- The researchers in [20] have conducted an experiment on a gamified unit testing process including two groups of individuals, gamified and non-gamified, and showed that the gamified group has had a significant outperformance in locating faults compared to the non-gamified group. Arnarsson and Johannesson in [21] reported that developers engaged in their experiment were motivated by their gamified system to create more effective unit tests and emphasized that the software testing skills of developers have been considerably improved. The authors of [22] introduced a game called "*Code Defenders*" that engages students in a competitive way to do mutation testing. In the game, the players can be defenders or attackers and SUT has a central role.

GWAP- In [24], a GWAP, called *Xylem, The Code of Plant*, was introduced for formal verification. In this game, the invariants of program loops are presented as strange plants. *Pipe Jam* is another game, introduced in [25], that carries out software verification by converting the task into a game puzzle and then converting back the solutions to a correctness proof. The first research to use GWAP in test data generation has been introduced in [13]. The game, called Rings, generates test data based on symbolic execution. In Rings, a CFG path is shown as a pipe network to players. At the network entry, there are several rings that fall in the network of pipes during the play. There are also some filters in the network. If the attributes of the rings are set correctly, the rings transit between the filters, successfully. The program conditional statements resemble the filters in the game and the input parameters of the source code map to the rings of the game. When the players solve the puzzles, they indeed generate data for the given source code. Although according to the evaluation results, it is successful as a GWAP, Rings has several problems, including:

– Mathematical complexity: Rings cannot visualize nonlinear and complex path constraints.
– Disposal of wrong solutions: if players are unsuccessful in solving a puzzle, all of their data are wasted, even if a wrong solution for the puzzle could be the correct solution for another puzzle.
– A large set of probable infeasible-test paths: Corresponding paths of puzzles that are not solved by players in the threshold time, are considered as a set of likely infeasible paths. However, a large set of likely infeasible paths is recommended by Rings while many of these paths are feasible.

In the present paper, we propose a new game which has the Rings' advantages while solves the above-mentioned problems, as well.

3 The Game Design of Greenify

The goal of Greenify is to generate test data based on the program's corresponding CFG with the aim of covering special test paths. To this end, the player somehow changes the input values to satisfy or dissatisfy the conditional statements of the program, implicitly, in order to cover a specific test path without any technical knowledge of the program variables, program control flow, and the conditions used in the branch statements.

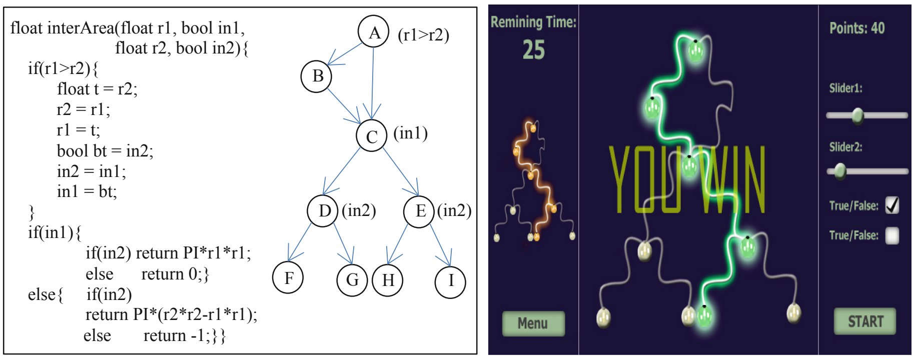
3.1 Display the Elements of a Program Unit in the Game

In this section, we explain how the components of a program (CFG and input variables) are displayed by game elements.

The Display of CFG in Greenify- The first step of designing Greenify is formulating CFG of SUT in a form of gameplay (or game environment). After extracting the CFG from the source code, it is displayed on the screen with some graphical appeal. Light bubbles that can present different colors are placed on the branches which are correspondent to the program's conditional statements. When a light bubble's color turns to green, it means that the corresponding conditional statement of the branch is evaluated as true, and when it turns to red, it means that the conditional statement is evaluated as false. For each level of the game, the player is asked to solve a path in the CFG.

The Display of Variables in Greenify- The SUT's input variables are mapped into sliders (for Integers, Floats and ASCII codes of characters) and checkboxes (for Booleans). The sliders are appropriate for both continuous and discrete variables. A player can change the input values by twiddling the sliders or checking and unchecking the checkboxes, without getting involved with the actual values of the variables.

For example, consider the code segment in Fig. 2 and its corresponding CFG. The goal of this program is to calculate the common area of two concentric circles. The two Boolean variables *in1* and *in2* determine if the outer or the inner area of each circle is intended in the calculation of the common area. Considering the CFG of this code, take for instance *ABCEH* as the target path of the graph that should be covered by appropriate input values. A screenshot of the game's environment for this situation is shown in Fig. 2 in which two sliders are shown for the two float inputs as well as two checkboxes for the two Boolean inputs.



```
float interArea(float r1, bool in1,
                    float r2, bool in2){
   if(r1>r2){
        float t = r2;
        r2 = r1;
        r1 = t;
        bool bt = in2;
        in2 = in1;
        in1 = bt;
   }
   if(in1){
             if(in2) return PI*r1*r1;
             else     return 0;}
   else{    if(in2)
             return PI*(r2*r2-r1*r1);
             else     return -1;}}
```

Fig. 2. The source code, CFG and game's environment of "interArea"

3.2 The Gameplay

A display of all described elements is shown to the player at the beginning of the game. The target path is shown to the player with blinking all the light bubbles of the path. The goal for the players is to adjust the input power flow by sliders or checkboxes to turn these light bubbles to green. At first, the player starts out just by randomly twiddling the sliders and checking the checkboxes. Each light bubble changes its color based on the corresponding values of the sliders and checkboxes. If a light bubble turns to green, this means the matching conditional is satisfied. However, if the bubble turns to red, the matching predicate is false (meaning that it is not covered by the corresponding input value). As the game goes by, the player can comprehend and learn patterns by viewing the results of his decisions and move the sliders and checkboxes in a more purposeful manner. Finally, when the player succeeds in turning all the light bubbles of the path to green, the level is passed and the player can move on to play other paths of the same graph or different graphs. The values with which the player completed the level are the generated test data resulting in the execution of the given test path for that level.

3.3 Logging the Events

As mentioned earlier, to complete each level of the game, all light bubbles in a specific path have to turn to green. However, during a play, some unwanted paths may be covered. When this happens, even though the level is incomplete, the values of the sliders may be appropriate test data for other test paths. These values can then be extracted as test data for those paths. This will not be told to the player since her target path is still uncovered. But, this feature is beneficial because test data can be produced for more than one test path in a single level. This way, a complete set of test data to achieve the coverage of all feasible test paths in a CFG is produced more quickly.

3.4 Special Cases in Test Data Generation with Greenify

The Inputs' Range: There is a challenge during the implementation of Greenify's sliders. The sliders, representing the input variables, have a limited length. Therefore, mapping an integer range to these sliders can be challenging. A solution considered in this article is to limit the input range in a more practical manner. For instance, in a program, it would be sufficient to consider only a part of the integer range. Based on the size of sliders on a mobile screen, we fitted the range from −10 to 10. As future work, we are planning to use another model instead of sliders, with which the player can change the precision simpler and choose the intended amounts, accordingly.

The Arrays: To simulate array data structures in the game, a new element is added to the game as shown in Fig. 3. If a player adds a new slider, indeed he adds a new element to the input array. The value of the array element is adjusted by changing the value of the new slider. Since the array in a program could have many elements, adjusting all elements for the players is very hard. Our solution to this challenge is generating data to all elements by random in the first time, and then the player could change each of the elements. To display a lot of sliders in the game, we use scroll bars such as Fig. 3.

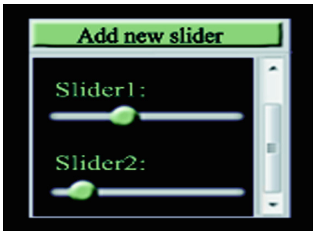


Fig. 3. A sample for considering program arrays in Greenify

Infeasible Test Path Detection: To guess infeasible test paths of the CFG, the approach used in this paper specifies a limited amount of time to complete a level (or cover a test path). If the player is not successful in that time period, she loses the level and another test path is given to her to cover. If the number of players who cannot solve the puzzle of a given path exceeds a specified threshold, the difficulty of that path increases and it will be given to higher-rated players. The more a path is given to players and is left unsolved, the higher the probability of it is infeasible.

Large CFGs: As the size of CFG increases, the game becomes harder to play. Since Greenify is merely designed to generate data for unit codes usually have small CFGs, large CFGs are not in our research scope. Nevertheless, to show the effectiveness of Greenify, in the next section, Triangle code is presented which has a large CFG with 57 paths such that their maximum length is 15. As the experiment showed, the players easily played the Triangle's game and did not engage in the apparent complexity of the game.

3.5 Players

Game flow and attracting players is important in Greenify. Therefore, in the following sub-sections, hinting and rating to players are discussed.

Hinting Players: It is important to design gameplay that is neither too easy nor too hard. Hints and clues could be used such that while the player is challenged, she doesn't lose hope and leave the game. A possible way to hint the player is to somehow show which light bubbles are affected when a slider is altered. For example, suppose the variable "A" is directly used in nodes labeled 15, 21, 23 and 30. So, if the player twiddles the slider representing "A", the corresponding light bubbles can blink, grow or change color. This way, the player is hinted which sliders should change to make a specific light bubble green. Furthermore, this would not make the game too easy because focusing on one light bubble to make it green can mess up the other light bubbles. Accordingly, the player has to use the hints wisely. To figure out which light bubbles are affected by each slider, we employed a scanner in our implementation to find the variables used in each program branch.

Rating Players: To make Greenify more attractive, players can be rated as they complete different levels. At first, players are rated as beginners, and they are asked to complete easier graphs with shorter test paths. As a player completes different levels, she gains more points and is rated higher. Players with higher rates are given harder and larger test paths. The difficulty of each test path is affected by the number of program branches (*b*), the number of the program's input variables (*i*) and the complexity of the program conditional statements in branches (*c*); the last parameter can be formulated as the number of variables or the number of clauses in a conditional statement. All of these parameters can be given proper weights by which the parameters' values are multiplied and averaged to come to a single value as the difficulty degree for a level. Equation 1 formulates the difficulty degree, were the corresponding weight for each parameter, *w1*, *w2*, and *w3* can be chosen by the game designer.

$$DifficultyDegree = \frac{b(w1) + i(w2) + c(w3)}{w1 + w2 + w3} \tag{1}$$

As an example, the *ABCEH* path of the graph shown in Fig. 2 has two branches, four input variables and the average branch complexity of 1.33 (The first branch's conditional statement includes two variables and the next two branches include 1 variable). If we respectively give a weight of *w1 = 2*, *w2 = 3* and *w3 = 1* to the parameters, the difficulty degree for this path would be 3.22.

3.6 Example: The Triangle Program

To better illustrate how Greenify works, an example is presented here. The example describes a game level based on a well-known program, called "Triangle", that by receiving three values as the three sides of a triangle, decides if the triangle is scalene, isosceles, equilateral or it is not a triangle at all. The source code of the program is shown in Fig. 4. This code has many infeasible test paths, and also its conditional statements are complex. Thus, many automatic test data generation approaches are unable to cover all feasible test paths.

The CFG of the "Triangle" program is illustrated in Fig. 5. The orange path, shown in Fig. 5, is the path that should be covered in this level of the game. In this path, it is determined that the triangle is isosceles. In Greenify, this path is first shown to the player. Then, by twiddling the three sliders, each corresponding to an input variable of the program, the player changes the input values. Each time the sliders are manipulated, the color of each node is changed, accordingly. For instance, if the player sets all the sliders to the same value, the path of the graph which leads to an equilateral triangle becomes green. Now, the player is supposed to play more with the sliders to turn the target path (shown in orange) to green. If she succeeds, she has finished the level and the values of the sliders are the test data generated as a result of completing this level.

```
private static int Triangle (int Side1, int Side2, int
Side3)
{
   int triOut;
   if (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)
     {  triOut = 4;
        return (triOut);}
   triOut = 0;
   if(Side1 == Side2)         triOut = triOut + 1;
   if(Side1 == Side3)         triOut = triOut + 2;
   if(Side2 == Side3)        triOut = triOut + 3;
   if (triOut == 0)
{
   if (Side1+Side2 <= Side3 || Side2+Side3<=  31
      Side1 || Side1+Side3 <= Side2)
                                  triOut = 4;
    else                          triOut = 1;
       return (triOut);}
   if (triOut > 3)         triOut = 3;
   else if (triOut == 1 && Side1+Side2 > Side3)
                                  triOut = 2;
   else if (triOut == 2 && Side1+Side3 > Side2)
                                  triOut = 2;
   else if (triOut == 3 && Side2+Side3 > Side1)
                                  triOut = 2;
   else        triOut = 4;
   return (triOut);}
```

Fig. 4. The source code of "Triangle"

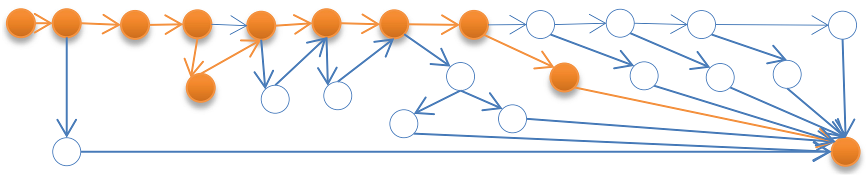


Fig. 5. CFG of "Triangle"

4 Evaluation

In this section, we compare the Greenify, with both Rings and conventional human-based test data generation, in terms of test data generation speed, coverage and proposed set of infeasible test paths. Since our purpose is improving the human-based approach, the automated approaches to test data generation are not in the scope of our research, and thus, we don't compare Greenify with these methods. Instead, we compare Greenify with random-based test data generation methods. In addition, we intend to evaluate the amount of entertainment of the game. So, the research questions are as follows:

1. Does Greenify generate test data faster in comparison to Rings, the conventional human-based and random-based approaches?
2. Is the path coverage of Greenify better in comparison to Rings, the conventional human-based and random-based approaches?
3. Does Greenify offer a smaller set for probable infeasible paths than Rings, the conventional human-based and random-based approaches?
4. Is Greenify more attractive than Rings for the players?

In the following subsections (Subsects. 4.1, 4.2, 4.3 and 4.4), answers to the questions 1 to 4 are described.

To conduct the experiment, simple versions of Greenify and Rings were designed for six benchmark programs (the benchmark programs are described in Table 1) and were given to 30 Computer Engineering students from the Faculty of Computer Science and Engineering of Shahid Beheshti University to perform a trial run. The average age of the participants was 21 years, 14 volunteers of them had programming experience, and 3 volunteers were familiar with concepts of software testing. The volunteers were divided into two groups each with 15 players. The players of the first group, first solved the Greenify puzzles, and after completing Greenify, they started to solve the puzzles of Rings; the participants of the second group solved the puzzles of the games in respectively reverse.

We also gave the six mentioned programs to five programmers and asked them to find input values manually, in a way to cover all the feasible paths of the code. In the end, we generated data for mentioned programs by the random-based test data generation approach. With the acquired data, we examined the amount of time consumed by the programmers, by the players in both games, and by the random method. We also computed the percentage of test paths covered by the test data generated via each method.

Table 1. The benchmark programs

#	Name	Description	Number of test paths
1	InterArea	The common area between two centric circles (Fig. 2)	8
2	Triangle	Determining type of a triangle (explained in Sect. 3)	57
3	Simple-Triangle	Determining type of a triangle	4
4	Binary-Search	Binary search algorithm	5
5	LCM	Determining least common multiplier	6
6	Reminder	Determining (x mod y)	4

4.1 Test Data Generation Time

According to the first research question, we compared Greenify with Rings, the manual and random approaches in terms of test data generation time.

The six benchmark programs contain 84 test paths, altogether, among which only 37 test paths are feasible. The total attempts of all the players in playing Greenify was 4263 times during which numerous test data was produced for all feasible test paths. In four competitors, the time taken for each feasible test path to be covered for the first time by any player was calculated in seconds, as displayed in Fig. 6. As shown in the figure, in the worst case, Greenify took 62.6275 s to complete a path. It is interesting that all other feasible paths were covered in less than 15 s. In the best case, it took 3.2250 s to complete a path. Both the best and the worst cases belong to the "Triangle" program. The path with the highest coverage time is the one through which it is determined that two of the inputs are equal, but the triangle inequality property is not held, and therefore, the three inputs cannot make out a valid triangle. The path with the least coverage time is the one through which it is concluded that at least one of the inputs is less than or equal to zero, and thus, the inputs cannot build a valid triangle. It is worth noting that some feasible paths were not covered by Rings and the random-based approach. The first time which players covered all paths for each program in four methods is shown in Table 2.

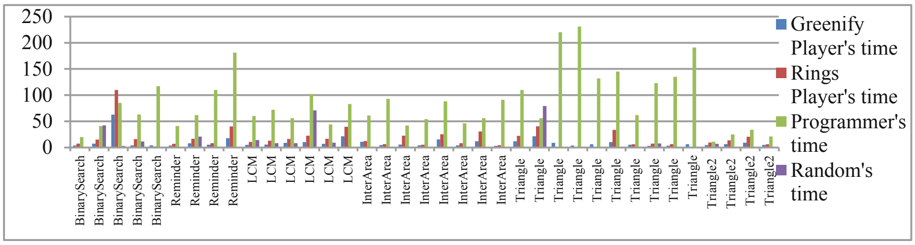


Fig. 6. Elapsed time (in seconds) for the first coverage of each feasible test path in Greenify, Rings, human-based and random testing

Table 2. Comparison of the first time that all paths are covered by the four test data generation approaches

	In Greenify (seconds)	In Rings (seconds)	By humans (seconds)	By random approach (the threshold time is 120 s)
InterArea	15.30	25.35	91	0.1
Triangle	21.25	40.34 (only 6 paths are covered)	231	120 (only 3 paths are covered)
simpleTriangle	9.03	20.454	34	120 (only 3 paths are covered)
Binary-Search	62.62	110.03	117	120 (only 4 paths are covered)
LCM	21.38	39.57	102	44.63
Reminder	17.90	40.214	181	7.74

As shown in Table 2, Greenify, Rings and the random-based approach have generated data for feasible paths faster than the human-based approach while Greenify outperformed Rings and the random-based method. The main reason for the better performance of Greenify compared with Rings is the use of wrong solutions obtained from Greenify players, which caused many paths to be covered by Greenify before they were proposed to the players. The random-based approach has generated data in less time for the two benchmarks while the speed of test data generation was less for other benchmarks compared to Greenify. In other words, Greenify generated data in less time for 66% of benchmarks.

To statistically analyze the time measured for covering paths by the four competitors (i.e., Greenify, Rings, human-based and random testing), we used the Anova test (Single Factor) method. A well-known test to compare the averages of two samples is T-test; but in case of more than two samples, it may be unreliable. Therefore, the Anova test could be a good choice to statistically analyze the time, elapsed to cover the paths, by the four competitors. By the Anova test, we can check whether there is a difference between the samples and it does not say which sample is better. Therefore, in the null hypothesis we indicate the averages of the samples are the same, and the alternative hypothesis is that the averages are different. We used the Anova test method in Excel Analysis Toolpak to compute all the needed data. In the results, F, F-critical, and P-value are equal to 46.64, 2.67 and 1.42E−20, respectively. Since F > Fcritical and P-value is less than the chosen significance level (P-value < 0.05), the null hypothesis is rejected. We compared the average elapsed times of Greenify and three other methods which seems to indicate that Greenify outperforms the competitors (Fig. 7).

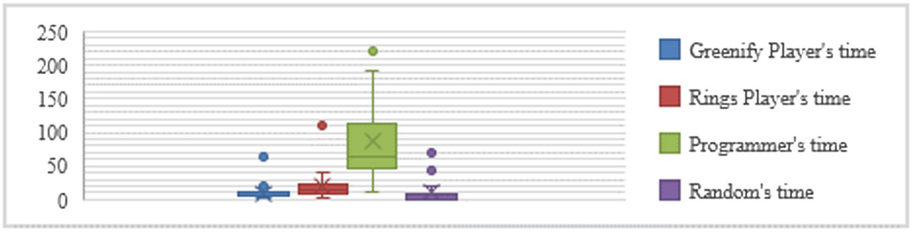


Fig. 7. Elapsed time comparison between the four test data generation approaches (Average times of Greenify, Rings, Programmers and Random are equal to 9.18, 18.83, 85.51 and 10.07, respectively)

Since the Anova test could not show Greenify has less average time than the three other methods, we use T-test to show that the difference between the average of the elapsed time of Greenify and the programmers is significant. The null hypothesis is "The average time of the programmers is less than the average time of Greenify". The calculated p-value is 2.58E−10. Since the p-value is less than the chosen significance level (0.05), the null hypothesis is rejected. Therefore, the average elapsed time for Greenify is less in comparison to the programmers.

4.2 Degree of Path Coverage

Based on the second research question, Greenify is compared with other approaches in terms of the path coverage criterion. Test data generated by Greenify and programmers covered all feasible paths. Rings covered 33 paths of 37 feasible paths and the random test data generation approach covered only 28 feasible test paths when the threshold time was 120 s. Feasible path coverage percentages are shown in Table 3.

Table 3. The results of path coverage percentage of the four competitors

	Greenify	Rings	Programmers	Random test data generation
Path coverage percentage	100%	89%	100%	75%

We used Kruskal-Walis test, a rank-based nonparametric test, to statistically analyze the number of covered paths of the benchmark programs by the four competitors. The null hypothesis in this analysis is "the number of covered paths for four categories is equal" and we want to see whether it is rejected. According to results, H and P-value are equal to 22.87 and 4.29E−5, respectively. Since H > P-value, the null hypothesis is rejected.

4.3 Estimation of Infeasible Paths

Based on the third research question, Greenify is evaluated in terms of the number of likely infeasible test paths. The “Triangle” program has 57 paths yet only 10 paths are feasible. This means that there is only a 17.5% chance for a player to be successful in completing the paths given to her. However, keep in mind that this program is a special case. In most other programs, the percentage of feasible paths is considerably higher.

Since the players have no idea whether a given level actually has an answer, they keep playing until they can win by passing as many levels as they can. If this problem was not handled by playing a game, developers obviously were not interested in spending this amount of time on it. Even if they were obligated to spend this time, they would not be satisfied with this part of their jobs. This somehow shows the engaging privileges of the game. Additionally, while the players in our experiment continuously tried to cover infeasible paths and failed, they unintentionally covered other paths by setting the game sliders to the input values needed to cover those paths. This way, even when they failed to complete the target path, they generated data for other feasible paths, and this scenario resulted in faster coverage of all feasible paths.

The players made 1050 attempts for the “Triangle” program and were able to find test data for 10 out of 57 test paths in the given time period set as the threshold for this game. Therefore, based on the idea mentioned, one can conclude that 47 paths of the “Triangle” program are most likely to be infeasible. The analysis of this program showed that these 47 paths were the exact set of existing infeasible paths of this program. On the other hand, Ring’s players were only able to cover 6 test paths of all the 10 feasible test paths of the “Triangle”. This means that the set of probable infeasible paths, proposed by Rings, was 41.

The number of proposed infeasible paths by the random method is 56. The random method generated 110000000 data in 100 s for the “Triangle” code but only 4 paths were covered while Greenify covered all feasible paths only by 1050 data. It shows that the generated test data by the random approach are ineffective due to its blindness while the players were smarter to generate test data.

4.4 Players’ Viewpoints About Greenify

According to the last research question, we asked the players to present their viewpoints about Greenify and Rings.

To have a better understanding of the players’ viewpoint on the game, a questionnaire was given to each of them after they finished playing. We asked players to rate their level of agreement about two questions on a five-point scale (1 = not at all, 5 = very) to compare the Rings and Greenify games in terms of their difficulty and enjoyability. Also, we asked them to choose a design goal to Greenify.

In Table 4 the average rates given by the players to both questions are shown. We use a statistical test (T-test) to show that difference between average rates of Greenify and Rings is significant. We selected the significance level of 0.05 and used the T-test function of Excel Analysis Toolpak to reject the null hypotheses shown in Table 4. The calculated p-values for both questions are shown in Table 4. Since these p-values are less than the chosen significance level, the null hypotheses are rejected. Therefore, the players chose Greenify, a more enjoyable and less complicated game to play.

Table 4. The result of players' answers to the questionnaire regarding enjoyability and difficulty of Greenify and Rings

Questions	Average rate to Greenify	Average rate to Rings	T-test parameters		
			Null hypothesis	Alternative hypothesis	P-value
1. The game was enjoyable.	3.86	2.63	Ring is more enjoyable	Greenify is more enjoyable	0.00053
2. The game was difficult to play	1.8	2.76	Playing Greenify is harder	Playing Rings is harder	0.00039

The interesting point in this survey was that even though the players were all students of computer engineering, only one of them guessed that Greenify had something to do with software quality or testing, and most of them thought it was designed just for entertainment (Fig. 8). The importance of this finding is that no basic knowledge is needed to play this game and anyone familiar with video games can easily play it.

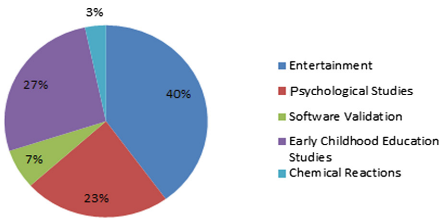


Fig. 8. Players' opinion about the design purpose of Greenify

5 Conclusions and Future Work

Based on the idea, explained in this paper, a game was designed and employed to generate test data. The main advantage of this idea is the use of inexpensive and copious agents (the players) that results in a decrease in costs and an increase in the speed as well as coverage of test data generation.

The results of the evaluation indicate that the proposed game outperforms Rings, the conventional human-based approach and the random method. Furthermore, the proposed game identifies a smaller set of likely infeasible paths. At last, the results of the conducted experiment reveal that the players mention Greenify as an attractive game.

As a direction for future work, we plan to provide more and larger benchmarks for better evaluation of Greenify. Another issue is finding a way to map the program's big input numbers into sliders. The solution we are considering is using another component instead of sliders, with which the player can change the precision, simpler, and choose the intended amount, accordingly.

A further issue that is worth considering is that the data collected from the behavior of the players as they attempt to cover paths can be very valuable for search-based methods. For instance, this data can be given to an appropriate learner and the behavior of the players can be analyzed to find further test data; therefore, the result can be a system for automatic test data generation. Because of the vast amount of data extracted from different players, a more powerful learner with higher precision and performance can be designed.

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, New York (2016)
2. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. **22**(5), 297–312 (2012)
3. Chen, T.Y., Fei-Ching, K., Robert, G.M., Tse, T.H.: Adaptive random testing: the art of test case diversity. J. Syst. Softw. **83**(1), 60–66 (2010)
4. Cadar, C., et al.: Symbolic execution for software testing in practice: preliminary assessment. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 1066–1071. ACM (2011)
5. Harman, M., McMinn, P., de Souza, J.T., Yoo, S.: Search based software engineering: techniques, taxonomy, tutorial. In: Meyer, B., Nordio, M. (eds.) LASER 2008-2010. LNCS, vol. 7007, pp. 1–59. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-25231-0_1
6. Weinstein, A.M.: Computer and video game addiction—a comparison between game users and non-game users. Am. J. Drug Alcohol Abus. **36**(5), 268–276 (2010)
7. Wightman, D.: Crowdsourcing human-based computation. In: Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries. ACM (2010)
8. Gong, D., Yao, X.: Automatic detection of infeasible paths in software testing. IET Softw. **4**(5), 361–370 (2010)
9. Werbach, K., Hunter, D.: For the win: How Game Thinking Can Revolutionize Your Business. Wharton Digital Press, Philadelphia (2012)
10. Deterding, S., Dixon, D., Khaled, R., Nacke. L.: From game design elements to gamefulness: defining gamification. In: Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments, pp. 9–15. ACM (2011)
11. Yuen, M.C., King, I., Leung. K.S.: A survey of crowdsourcing systems. In: 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom). IEEE (2011)
12. King, J.C.: symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
13. AmiriChimeh, S., Haghighi, H., Vahidi-Asl, M., Setayesh-Ghajar, K., Gholami-Ghavamabad, F.: Rings: a game with a purpose for test data generation. Interact. Comput **30**, 1–30 (2017)
14. Mao, K., Capra, L., Harman, M., Jia, Y.: A survey of the use of crowdsourcing in software engineering. J. Syst. Softw. **126**, 57–84 (2017)
15. Schmitz, B., Felicia, P., Bignami, F.: An international survey In: Gamification in Education: Breakthroughs in Research and Practice, pp. 439–452. IGI Global (2018)

16. Reeves, N., West, P., Simperl, E.: "A game without competition is hardly a game": the impact of competitions on player activity in a human computation game. In: AAAI (2018)
17. Prabhakar, N., Singhal, A., Bansal, A., Bhatia, V.: A literature survey of applications of meta-heuristic techniques in software testing. In: Hoda, M.N., Chauhan, N., Quadri, S.M.K., Srivastava, P.R. (eds.) Software Engineering. AISC, vol. 731, pp. 497–505. Springer, Singapore (2019). https://doi.org/10.1007/978-981-10-8848-3_47
18. Baker, A., Navarro, E.O., Van Der Hoek, A.: An experimental card game for teaching software engineering processes. J. Syst. Softw. **75**(1), 3–16 (2005)
19. Sheth, S., Bell, J., Kaiser, G.: Halo (highly addictive, socially optimized) software engineering. In: 1st International Workshop on Games and Software Engineering, pp. 29–32 (2011)
20. Johansson, M., Ivarsson, E.: An experiment on the effectiveness of unit testing when introducing gamification. Ph.D. thesis, Master's thesis, Chalmers University of Technology (2014)
21. Arnarsson, D., Johannesson, I.H.: Improving unit testing practices with the use of gamification (2015)
22. Rojas, J.M., Fraser, G.: Code defenders: a mutation testing game. In: 11th International Workshop on Mutation Analysis. IEEE (2015)
23. Navarro, E.O., van der Hoek, A.: SIMSE: an interactive simulation game for software engineering education. In: CATE, pp. 12–17 (2004)
24. Logas, H., et al.: Software verification games: Designing Xylem, The Code of Plants". In: FDG (2014)
25. Dietl, W., et al.: Verification games: making verification fun. In: 14th Workshop on Formal Techniques for Java-like Programs, pp. 42–49 (2012)
26. Gómez, M., et al.: Reproducing context-sensitive crashes of mobile apps using crowd-sourced monitoring. In: 2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft). IEEE (2016)
27. He, M., et al.: A crowdsourcing framework for detecting cross-browser issues in web application. In: Proceedings of the 7th Asia-Pacific Symposium on Internetware. ACM (2015)
28. Afzal, W., et al.: An experiment on the effectiveness and efficiency of exploratory testing. Empir. Softw. Eng. **20**(3), 844–878 (2015)
29. Leicht, N., et al.: When is crowdsourcing advantageous? The case of crowdsourced software testing (2016)
30. Schneider, C., Cheung, T.: The power of the crowd: performing usability testing using an on-demand workforce. In: Pooley, R., Coady, J., Schneider, C., Linger, H., Barry, C., Lang, M. (eds.) Information Systems Development, pp. 551–560. Springer, New York (2013). https://doi.org/10.1007/978-1-4614-4951-5_44
31. Gomide, V.H.M., et al.: Affective crowdsourcing applied to usability testing. Int. J. Comput. Sci. Inf. Technol. **5**(1), 575–579 (2014)