Scientia Iranica D (2023) 30(5), 1670–1686

Sharif University of Technology
Scientia Iranica
Transactions D: Computer Science & Engineering and Electrical Engineering
http://scientiairanica.sharif.edu

Test data generation for program units using a game with a purpose

Sh. Moosavi, M. Vahidi-Asl*, and H. Haghighi

Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran.

Received 6 April 2022; received in revised form 13 November 2022; accepted 11 April 2023

KEYWORDS
Software testing;
Test data generation;
Serious game;
Game with a purpose;
Human-based computation game.

Abstract. Today, several methods have been presented to automate test data generation; because of the low maturity level of automatic methods, this is still widely carried out by humans in industry. Hence, the challenge is finding approaches in which humans could generate test data through more attractive, faster, and cheaper ways. To achieve this, one approach is using a Game With A Purpose (GWAP) in test data generation. In our previous work, we introduced two games called Rings and Greenify, by which many inexpensive players with no special technical abilities become engaged in test data generation. This paper presents an entirely new GWAP for test data generation called Quest Of Treasure Explorer (QOTE). QOTE provides a different gameplay and has certain advantages compared to prior games, including faster generation of test data, easier puzzles, narration, etc. Experimental results have shown that QOTE outperforms prior games in two aspects: game quality and capability of test data generation. We have also conducted an experiment based on mutation analysis to further evaluate the test data generation capabilities of QOTE compared to four automatic approaches. According to this experiment, QOTE outperforms the four competitors regarding average mutation scores.

1. Introduction

Test data generation is one of the main but costly tasks in the overall process of software testing [1,2]. Since many software systems have numerous choices for input data during the testing process, test data generation methods aim to find as small as possible subsets of input values that result in more effective tests, which reveal a larger number of failures in the SUT.

Because of the importance of test data generation, a vast amount of research has been conducted to automate this activity [2,3]. Despite significant advances in automatic test data generation approaches, the majority of software companies prefer to employ manual techniques [4]. Generating test data by humans has advantages, among which is the ability of humans to understand and interpret the program under test. In addition, the computing power of a human mind helps him/her solve problems with a high level of complexity. However, manual test data generation is a tedious task, and the exhaustion of human agents during the process and their lack of motivation make the human-based method too time-consuming.

Since test data generation is still widely carried out manually, the challenge is finding faster, more attractive ways for humans to generate test data. A solution is to abstract away the test data generation process so that non-technical people can participate in the process. One way to achieve this objective is to transform the programs under test and the test data generation process into appropriate graphical views.

Today, due to the widespread use of social networks and intelligent devices, gaming has become a

*. *Corresponding author.*
E-mail address: mo_vahidi@sbu.ac.ir (M. Vahidi-Asl)

doi: 10.24200/sci.2023.60224.6673

universal practice [5,6]. The computing power of the human mind can be harnessed for an implicit purpose via playing the game. This problem-solving approach introduces the notion of the Game With A Purpose (GWAP) or Human-Based Computation Games (HBCG) [7,8]. We have investigated the applicability of human-based computation games in test data generation. As a first attempt, we restricted our research scope to the test data generation of program units. This resulted in two games, called Rings [9] and Greenify [10], which are reviewed in Section 2. We demonstrated that by using the notion of GWAP for test data generation, the main problems of human-based test data generation could be alleviated in this way because:

- The problem is presented in a non-technical form; so it can be given to a broader range of people to solve;
- The lack of motivation and fun among testers is replaced with the game's amusement;
- The cost of test data generation is reduced since many inexpensive people can contribute.

Although our two prior games successfully provided an entertaining interface for cheap test data generation, they had certain drawbacks and unresolved challenges described in Section 2. This paper introduces a new GWAP, Quest Of Treasure Explorer (QOTE), for test data generation. QOTE is proposed to address the issues of the prior games and also outgo their performance. Accordingly, the research questions of this paper are as follows:

- Question 1: How to design QOTE such that the players generate test data faster than when they use the prior games?
- Question 2: How to design QOTE such that it is more user-friendly and easier to interact with in comparison to the prior games?
- Question 3: How to design QOTE such that it is less complicated than the prior games? In other words, how to design QOTE such that: (1) The players need less analysis to solve their puzzles? (2) Could it be applied to programs with complex operations?
- Question 4: How can QOTE be designed to use players' wrong solutions?
- Question 5: How to design QOTE to be more enjoyable than the prior games?
- Question 6: How well does QOTE generate test data compared to the well-known automatic test data generation tools?

The remaining parts of this paper are organized as follows. In Section 2, the related works are reviewed. Section 3 presents the design and the mechanics of QOTE. In Section 4, QOTE is evaluated, and the experimental results are discussed in Section 5. Finally, in Section 6, the conclusion and directions for future work are presented.

2. Related work

A category of games is serious games where the purpose is beyond entertainment. In this category, the game design includes all the elements and details of a typical game [11]. Incorporating games with human-based computation has led to the emergence of GWAP, which constitute a subset of serious games [7]. Recently, serious games and GWAP have been used in various fields of software engineering. Some of these games are explained briefly in the following.

Pex4Fun is a web-based serious game to teach computer programming to computer students. It encourages the students to edit a given program code in different browsers, which are then given to the Pex4Fun engine to execute and analyze [12]. *Code Hunt* is an extension to *Pex4Fun* and a serious game platform for practicing programming skills. It is based on *Pex*, a symbolic white box execution engine [13].

Fava et al. [14] presented an approach to employing GWAP to improve the detection of program invariants. This approach transforms the task of program invariant detection into a computer game, the so-called *Binary Fission*, which is based on precondition mining. In this game, the player acts as a classification engine and detects invariants using filters in a graphical presentation.

Rojas and Fraser [15] introduced a game called *Code Defenders*, which uses game elements in the testing process. The game engages students in a competitive and fun way to perform mutation testing while adding fun to the learning experience. In the game, the players act as attackers or defenders; The attackers generate mutants in a source code, and the defenders create unit tests to kill the mutants.

Chen and Kim [16] proposed a Puzzle-based Automatic Testing environment (PAT) to decompose object mutation and complex constraint-solving problems into small puzzles for solving by humans. The evaluation results showed that humans could solve problems effectively. The proposed system was purely based on puzzle-solving, and the role of the game elements was not bold. Moreover, the game environment was more similar to a gamified system rather than GWAP.

The most similar approach that investigated the applicability of GWAP in test data generation, called Rings, was presented by Amiri-Chimeh et al. [9]. *Rings* aims at generating test data for program units based on symbolic execution. In each puzzle, a Control Flow Graph (CFG) path is displayed to the players by a

network of pipes. An entry node has several rings, which can fall into the pipe network. The pipes contain several filters. The filters are representations of the program constraints, and the rings resemble the input parameters of the source code. If the properties of the rings (e.g., size, shape, or color) are not set correctly, the rings cannot pass through the filters. When a player solves this puzzle, he/she is implicitly generating appropriate values to satisfy the corresponding path constraint. According to the evaluation results reported by Amiri-Chimeh et al. [9], Rings was successful as a GWAP, and its players have covered all paths of the benchmark programs. However, this game has several unresolved challenges, such as limitations of the user interface, mathematical complexity, long puzzles, disposal of wrong solutions, lack of narration, and a large set of likely infeasible test paths.

In Greenify [10], a prior version of the game is going to be presented in this paper, the paths of a unit's CFG are represented through a string of connected light bubbles, and the input parameters of the unit are displayed via sliders and checkboxes. The players should change the values of the sliders and checkboxes until the color of all connected bubbles turns green. The results indicated that Greenify outperformed Rings, the conventional human-based approach, and the random test data generation method in terms of path coverage percentage and elapsed time for generating test data [10]. However, Greenify could not solve the following challenges of Rings:

- Limitations of the user interface;
- The lack of narration.

In addition, Greenify has unsolvable challenges including the demonstration of wide-range input parameters through sliders, the presentation of impossible levels to players, which frustrates them, and scalability, where the challenge is the limitation of the test path length.

In this paper, we are going to design a new game based on Greenify, which has the advantages of Rings and Greenify while handling their above-mentioned drawbacks.

3. Game design

Our approach has three phases (Figure 1). In the first phase, unit codes are preprocessed, and their corresponding CFGs are prepared for game generation. In the second phase, the most important phase in the game design, game levels are automatically generated, and game puzzles are created based on the unit codes' CFGs. In the third phase, the game is played by several players, and then the generated data is collected and refined; finally, the resultant data set is presented as test data of the corresponding program unit. The first and second phases are demonstrated in the following, but the third phase is out of scope of this paper.

3.1. First phase: Program preprocessing

In this phase, the program units under test are scanned, and if they contain loops, the code should be transformed into an appropriate form. After code transformation, the CFG of the unit code is created by the CFG class of Clang, and subsequently, the input domain of the source code is reduced.

3.1.1. Transforming code

Loop constructs in programs may introduce an infinite number of paths. Some approaches have been suggested to deal with loops. For example, a typical testing method is to choose test paths that skip the loop, repeat the loop exactly once, and iterate the loop more than once [1]. Similarly, in this paper, three possibilities for loops are considered: zero, one, and more than one repetition. For this purpose, we introduce three conditional statements for each loop to simulate the three mentioned possibilities. Also, a variable named *Loop Repeat* is defined for each loop, which is incremented in the loop body to count the number of loop iterations. Figure 2 shows an example of how a loop is transformed into an *if* conditional statement. The original code and the corresponding CFG are shown in Figure 2(a). Figure 2(b) shows the code after transformation and the corresponding CFG. Having this CFG in hand, we aim to generate data to cover three new paths (ABE, ABCE, ABCDE) in the CFG of Figure 2(b) instead of all paths in the CFG of

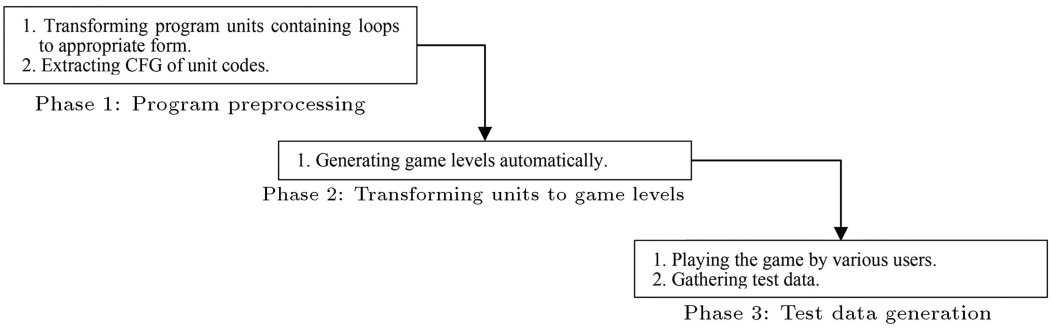
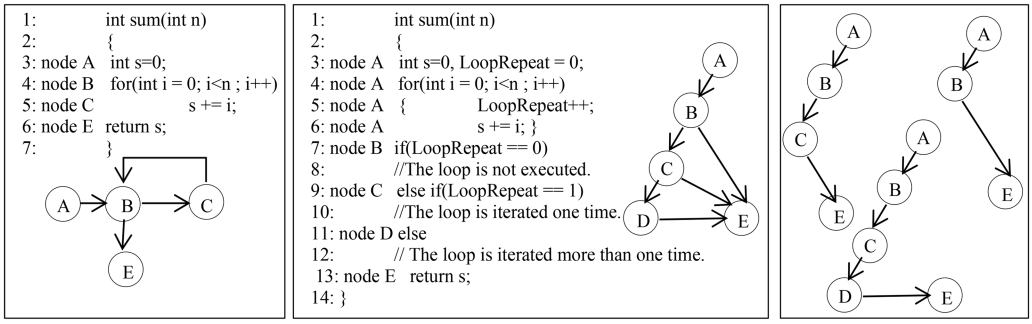


Figure 1. Three phases of the QOTE's game design.



(a) The main code and corresponding CFG

(b) The changed code and the corresponding modified CFG

(c) The modified CFG is then decomposed to three sub-CFGs

Figure 2. An example of modifying the unit code and CFG.

Figure 2(a) (i.e., ABE, ABCBE, ABCBCBE, ...).

3.1.2. Extracting CFG

We use the CFG class of the Clang tool to extract the CFG of a given unit code. Since our proposed game requires that each path is specified, distinctively from other paths to the players, each program path should end with a unique exit node. To discriminate the included test paths, the modified CFG is decomposed into several sub-CFGs such that each path ends with a unique exit node. In Figure 2(b), the CFG of the given program has three test paths with a common exit node, so it should be decomposed into three sub-CFGs, as shown in Figure 2(c).

3.2. Second phase: Automatic game level generation

This section is devoted to the description of QOTE's design.

3.2.1. Puzzle design

Each level of QOTE happens in an old house. In each old house, there might be many safety boxes. Each safety box is connected to its own "gear train". A gear train is a sequence of gears and chains, which is started from the lock system (explained further) and is terminated at a specific safety box. Different gear trains in a house make a "system of gears". In other words, a system of gears is a graph of gears where each node is represented by a specific gear, and two connected nodes are represented by two gears, which are chained together. In the upper part of the system of gears, there exists a system of locks (including several types of locks). Therefore, a puzzle in a given old house includes several gears and chains (system of gears), several locks (system of locks), and multiple safety boxes (Figure 3(b)).

When a player enters an old house, the whole gear system is located on the ground. The player should dial the locks to find the passcode of a safety box. Based on the values of the locks, some "on-the-ground" gears and chains might rise to a certain height above the ground, and the gears begin to spin. These "above-the-ground" gears and chains are the ones that can transmit power to their neighboring gears. A correct passcode turns all of the "on-the-ground" gears and chains to "above-the-ground" transmitting rotation and power from the locks to the corresponding safety box and consequently opening it.

- ***Relationship between the game design and test data generation:*** Test data generation in software testing aims to find appropriate values for input parameters of a source code to satisfy a specific test criterion. We selected the path coverage criterion for QOTE. The process of dialing the locks to unlock a connected safety box corresponds to the process of test data generation for a given test path of a program under test.

  Each region of the land in QOTE corresponds to a program where the old houses of that region represent the units of that program (Figure 3(a)). A safety box and its connected gear train resemble a desired CFG test path for which we wish to generate test data. The game is designed such that finding a passcode of a given safety box is equivalent to generating input data that covers the corresponding CFG test path. This indicates that discovering all the games in a house leads to the generation of test data for all the targeted test paths of the corresponding CFG.

- ***Designing the game according to program units:*** The game design elements and their correspondence to the CFG and the input parameters of the program are as follows:

  • The display of CFG in QOTE: The entry and intermediate nodes of the CFG are presented by simple gears, and each final node is displayed by a safety box. An edge in the CFG is illustrated by a chain. If the number of CFG's test paths is abundant, only some of the test paths are displayed to the player at a game level, and the rest is displayed at other levels. The maximum

(a) Outdoor environment of the game

(b) Indoor environment of the game

Figure 3. Two snapshots of QOTE.

(a) Keyboard for character data type

(b) Switch for boolean data type

(c) Safe combination lock for numeric data type

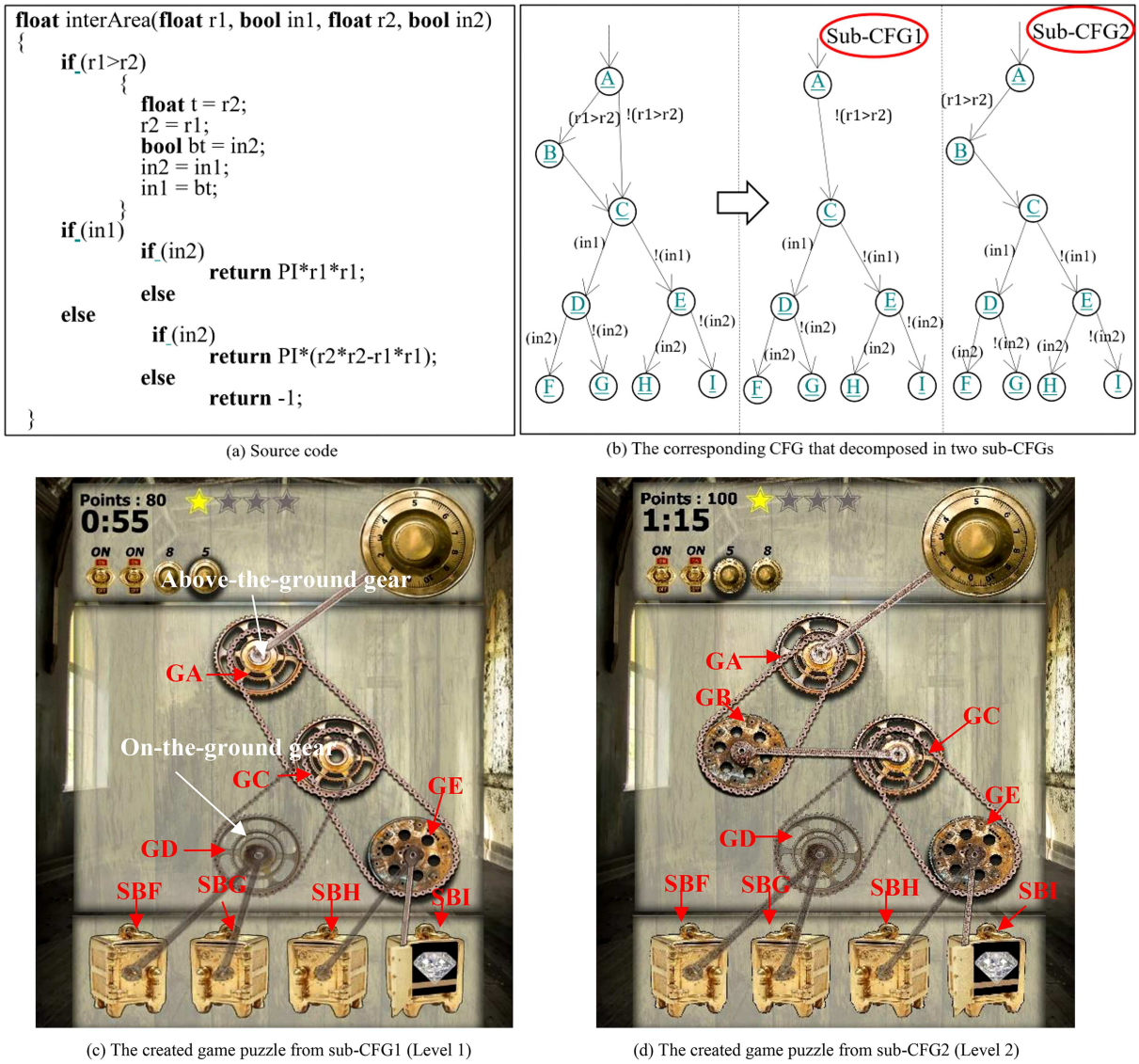
Figure 4. Different types of locks in QOTE.

number of displayed test paths in a game level is determined by the game designer;

• The display of the CFG paths in QOTE: A path in a CFG is represented by a gear train ending in a safety box. Each time the player dials the locks, the generated values are given to the corresponding SUT, which is then executed with those values. Whenever the SUT execution reaches a conditional statement, represented by a gear in the specified train, the gear rises to a certain height above the ground and begins to spin. This way, the player learns that the given input value has affected the gear;

• The display of SUT's input parameters in QOTE: The input parameters of the SUT are mapped to the system of locks. A system of locks is a combination of several locks designed according to the data types of the SUT's input parameters. In this version of the QOTE's design, various data types are supported, as shown in Figure 4.

– ***An illustrative example:*** Consider the code segment in Figure 5(a) with the corresponding CFGs in Figure 5(b) (the original CFG is decomposed into two sub-CFGs, each sub-CFG is shown in a separate game level). This program aims to calculate the common area of two concentric circles. Two Boolean variables, *in1*, and *in2*, determine if the outer or the inner area of each circle should be included in the calculation of the common area. A screenshot of the two game's levels for this unit is shown in Figure 5(c) and Figure 5(d). As demonstrated in



(a) Source code

(b) The corresponding CFG that decomposed in two sub-CFGs

(c) The created game puzzle from sub-CFG1 (Level 1)

(d) The created game puzzle from sub-CFG2 (Level 2)

Figure 5. An example of a unit code, the corresponding CFG, and QOTE puzzles.

Figure 5(c), there are two combination locks and two switches for the two Floating-point inputs and the two Boolean inputs. The intermediate nodes A, C, D, and E are represented by four gears GA, GC, GD, and GE, respectively. Since A is connected to C in the CFG, the gear GA is connected to the gear GC by a chain. Similarly, since the nodes D and E are the left and the right child of C, the gears GD and GE are connected to the left down and the right down corner of the gear GC, respectively. The final nodes, F, G, H, and I, are represented by four safety boxes SBF, SBG, SBH, and SBI, respectively. As can be seen, these safety boxes are connected to gears, corresponding to their parents in the CFG. Accordingly, two safety boxes SBF and SBG, are connected to GD. Similarly, two safety boxes, SBH and SBI, are connected to GE.

Considering the CFG of the code, let's take the path called ACEI, which includes A, C, E, and I nodes, as the target path of the graph. As can be seen, the CFG path ACEI is represented by a gear train of GA, GC, and GE, which is connected to SBI. To open SBI, the player has to dial four locks so that GA, GC, and GE move above the ground (these gears are shown in lighter colors in Figure 5(c)). As mentioned earlier, gears rise above the ground when their corresponding conditional statements are satisfied. This means that by opening SBI, the players find values that make the conditional statements $!(r1 > r2)$, in1, and in2 become true, making the program's execution follow ACEI.

3.2.2. Real-world challenges of using QOTE

In this section, the main challenges and our solutions to handle them are described:

(a) (b)

Figure 6. Safe combination locks.

- ***The number of input parameters:*** If a program has many input parameters, the lock system cannot be properly visualized because the number of locks should be proportional to the screen size. To solve this challenge, we use small icons shown below of the system of locks. If the player clicks on one icon, the corresponding lock in real size appears on the system of locks (the upper side of Figure 5(c)).
- ***Designing safe combination locks for numerical data types:*** The numeric data types (integer or Float) have a wide range of values. Therefore, the safe combination lock should be susceptible to changes by the player. To alleviate the challenge, we could use two or more dialers in a combination lock. Each dialer represents two digits of the parameter value. For example, in a combination lock with two dialers, the first dialer may represent the ones and tens digits of a natural number; the second dialer may represent the hundreds and thousands of digits of the number (Figure 6(a)).
- ***Scalability problems:*** Scalability in QOTE is questionable in two aspects:
  - The abundant number of paths: If the number of CFG paths is more than a specific value (more than five paths based on our observations in experiments), playing the corresponding game level on a mobile screen could be challenging for players;
  - The long test paths: Another significant issue is related to the test path length. Although a puzzle may have a long gear train, based on our observations in the experiments, following a path with more than ten nodes by a player in a limited time might be difficult.

In QOTE, we can easily decompose a level with abundant paths into several levels with the normal number of paths. Therefore, the first aspect of the scalability problem is solved; our solution to the second aspect of the scalability problem is as follows.

We first sampled to estimate the percentage of real-world unit codes having paths with more than ten nodes in their corresponding CFG. In this experiment, a sample of 600 code functions was selected from the repositories of GitHub. We checked the CFG of the methods and determined the length of their longest path. The results show:

- In 90% of the unit codes, the length of the longest CFG paths is no more than ten;
- The average length of the longest CFG paths of all existing unit codes is 4.54.

The results indicate that unit codes usually do not have long paths, and the scalability problem is not challenging for them. Nonetheless, in QOTE, a solution is designed to handle long paths. In the proposed solution, a long path is divided into several short paths, and then, the union of short paths is displayed to the players. At the end of the gameplay, the intersection of all generated data for short paths could be considered as test data covering the original long path.

For example, the code for determining the type of a triangle and its CFG are displayed in Figures 7 and 8, respectively. To display how to generate test data for a long path, consider path ABCDMENFOGHQZ with 13 nodes corresponding to an equilateral triangle. Due to the path length, it is divided into two shorter paths, as shown in Figure 9. The two shorter paths are further connected by a new node, the red node, in Figure 9. Some generated data to cover sub-paths ABCDME and NFOGHQZ are $\{\{1,1,1\}, \{1,1,2\}, \{2,2,2\}, \{2,2,1\}, \{3,3,1\}\}$ and $\{\{1,1,1\}, \{2,2,2\}, \{3,3,3\}\}$, respectively. The intersection of two sets is $\{\{1,1,1\}, \{2,2,2\}\}$ that covers the original long path.

```
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 <= 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);}
```

Figure 7. The source code of the triangle example.

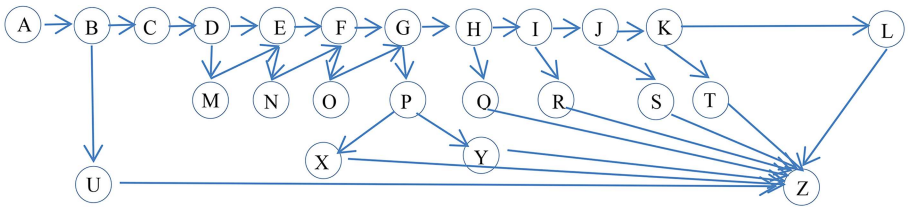


Figure 8. The CFG of the code.

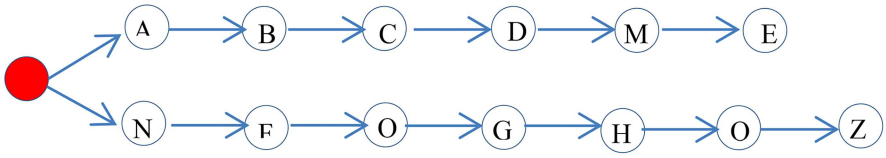


Figure 9. The changed long path of the CFG.

Figure 10. New element to simulate arrays in the game.

To show how the scalability problem is solved in QOTE, we have conducted several experiments on some large benchmark units.

– ***Non-primitive data types:*** In addition to simple data types (such as integer, float, char, etc.), there are non-primitive data types (like array, string, list, etc.) in unit codes. We proposed a solution to handle these data types in QOTE. In this version of the game, we merely provided a solution for arrays. In this version of the game, two types of arrays are considered:

1. The array size is fixed, but it is larger than ten: Our solution to this challenge is generating data for such arrays by an automatic approach presented by Michael et al. [17].
2. Dynamic-length or fixed-length arrays with a size less than ten: In this situation, to simulate arrays, as was done in Greenify, a new element is added to the game, as shown in Figure 10. This element allows the players to add a new lock/switch/keyboard to the game. The details of our solution to simulate arrays have been presented by Moosavi et al. [10].

– ***Infeasible test path detection:*** Finding infeasible paths of a given program is a challenging problem [1]. To identify infeasible test paths of a given CFG, like in Greenify, we consider a time limitation for each player to complete a level (i.e., cover a test path). If a path that is given to various players has not been covered by any player in the specified time and is left unsolved, it is more likely to be an infeasible path. A set of likely infeasible test paths is recommended by QOTE. We assume that these paths are reviewed by programmers to determine which of them is infeasible.

4. Evaluation

In the previous section, we introduced QOTE as a new GWAP for test data generation. In this section, to evaluate the performance of QOTE, we have conducted two experiments. The comparison of QOTE with the prior games is presented in Section 4.1. In Section 4.2, we compared QOTE with four automatic approaches.

4.1. Experiment 1: Comparison of QOTE with the prior games

4.1.1. Part 1: Comparison of QOTE with Rings

Experimental setup: First, we implemented Rings and QOTE so that the puzzles of the games could be automatically generated from the CFG of a given program unit. The games were developed by the Unity3D game engine, Version 5.1. The graphics of the games were made using Adobe Photoshop CS6.

Benchmarks: To conduct comparisons, we selected some standard C++ programs (Table 1), typically used

Table 1. Benchmark programs.

Program number	Program name	Program description	Lines of code	Number of functions	Number of test paths
1	Simple triangle	Determining the type of a triangle	32	1	4
2	Triangle	Determining the type of a triangle with an algorithm different from Simple Triangle in which the number of infeasible test paths in the corresponding CFG is abundant (the program has 57 paths, yet only ten paths are feasible)	40	1	57
3	Inter area	Determining the common area of two concentric circles	29	1	8
4	Power	Determining the value of expression x^y	19	1	4
5	Reminder	Determining the reminder of expression x/y	30	1	4
6	Binary search	Searching a number as a key in an array of numbers	55	1	11
7	LCM	Determining the least common multiple of two numbers	24	1	6
8	Roots quadratic equation	Determining the roots (both real and complex roots depending upon the discriminant) of a quadratic equation	41	1	4
9	Two squares	Determining the intersection of two squares (user is given two squares, the sides of one square are parallel to the coordinate axes, and the sides of the other are at 45 degrees to the coordinate axes)	29	1	3
10	Bessj	Determining the value Bessel J_n	245	3	33
11	Expint	Determining the value of the exponential integral	109	1	38
12	Fisher	Determining fisher statistical	157	1	183
13	Gamma	Gamma function	112	4	27
14	Comput tax	Computing the amount of tax	164	1	24
15	Line	Determining the situation of four lines relative to each other	106	1	18
16	Print calender	Print calendar according to the input of year and month	187	9	53
17	Tcas	Aircraft avoid collision system	199	9	27

in test data generation research [1,2], and extracted 38 functions from them as benchmarks.

Rings was not able to create puzzles for 310 out of 504 test paths of the benchmark programs due to the existing "complex operators" compared with QOTE, which managed to generate puzzles for all the test paths. We packaged 194 (i.e., 504 − 310) Rings and QOTE puzzles into two packages, called "MainRings" and "MainQOTE", respectively. The puzzles of "MainQOTE" and "MainRings" are organized into 21 and 194 levels, respectively. Finally, we packaged the remaining 310 QOTE puzzles into the "ExtraQOTE" package, including 32 levels. It is worth mentioning that among 504 test paths, 222 paths were infeasible.

Participants: The evaluation involved a sample of 50 participants, including 23 women and 27 men. The participants were students of an Engineering Mathematics Class studying different engineering fields. We excluded candidates that were studying computer engineering or computer science. This does not mean people with skills in these fields are not potential players. We only ignored them from the study to guarantee the playability of QOTE for non-technical individuals. The mean age of volunteers was 19 years (from 18 to 22), and their mean education years were 13 (range of 11 to 15).

To decrease the effect of the chance factor on the results, we asked the individuals to participate in four sessions (one session a week) and perform similar tasks in all sessions.

In each session, these 50 players were randomly divided into two groups, each with 25 players, called "RingsFirst" and "QOTEFirst", considering which game they started playing at first. Since "MainRings" and "MainQOTE" have been designed from similar test paths, participants who have solved a puzzle in one of the games may solve the corresponding puzzle in the other game more quickly. With this grouping, we could compare the results more fairly.

Procedure: The procedure of sessions includes the following steps:

1. In the first session, players were briefed about the research;
2. In all the sessions, the executable file of "MainRings" (in the APK format) was handed to the members of "RingsFirst" to install it on their smart devices and then start playing simultaneously. The same was done for "MainQOTE" and the "QOTEFirst" players;
3. At each stage, game levels with equal difficulty degrees were randomly distributed among the players to give the players a random chance to play at different levels. Furthermore, the solved puzzles by wrong solutions of other levels are not suggested to the players;
4. For each player who stopped playing one of the games, we gave him/her the executable file of the other game and asked him/her to play it, as well;
5. When a player stopped playing the second game, we asked each player to play "ExtraQOTE";
6. In the last session, we asked each player to fill out a printed questionnaire (Figure 11).

Evaluation Criteria and Metrics: Our evaluation is based on the criteria in Table 2. These criteria have been chosen according to our research questions.

4.1.2. Part 2: Comparison of QOTE with Greenify

The methodology and experiments to compare QOTE with Greenify were similar to what was explained in the previous subsection. The details are described below:

- The benchmarks introduced in Table 1 were also used in this experiment. Since QOTE and Greenify create similar puzzles, they are packed into two packages (say "MainQOTE" and "MainGreenify");
- The participants in this experiment were similar to those in the previous experiment: a sample of 50 students, including 23 women and 27 men, who attended a Physics Class. The players were randomly divided into two groups, "QOTEFirst" and "GreenifyFirst", considering which game they started playing at first;
- The procedure of each session in the experiment included similar steps to the previous subsection;
- The same evaluation criteria and metrics as the previous section were selected.

4.1.3. Results

In this section, we present the results of the conducted experiment. We summarize the collected results in Tables 3, 4, 5, and 6.

4.2. Expriment 2: Comparison of QOTE with four automatic approaches

After comparing the prior games, we selected four well-known automatic tools to generate test cases: Pex, Evosuite, Fuzz testing, and Pseudo-Random.

QOTE and the four selected tools have different coverage criteria. A well-known approach to compare the effectiveness of test data generation techniques with different coverage criteria is mutation analysis [18,19].

The mutation analysis on test suites generated by the competitors is performed by the PIT tool, which is the main reference tool for performing mutation analysis [20].

QOTE outperforms the four competitors in terms of average mutation scores: 88% (QOTE) versus 62%

A. Personal information

Age:

Gender:

Years of education:

B. Game experiences

Questions	Your rate to QOTE (1 to 5)	Your rate to Rings (1 to 5)
1. The game was engaging.		
2. The game was boring.		
3. I would like to continue playing the game.		
4. While I play the game, I forgot about where I was.		
5. The game was fun		
6. The game's user interface was easy to use.		
7. Playing the game was hard.		

Figure 11. The questionnaire.

Table 2. The criteria and metrics.

Criteria	Metric
The needed time to generate data by each game (concerning research Question 1).	The time needed by the "RingsFirst" and the "QOTEFirst" players to collectively solve the corresponding "MainRings" and "MainQOTE" puzzles, respectively.
Easy interaction with the interface of the games (concerning research Question 2).	The percentage of the participants who selected QOTE as their answer to the sixth question of the questionnaire (Figure 11).
The complexity of the games' puzzles (concerning research Question 3).	The percentage of the participants who selected QOTE as the answer to the seventh question of the questionnaire (Figure 11).
The capability of the games to use players' wrong solutions for test data generation (concerning research Question 4).	The percentage of the QOTE puzzles which were solved by a wrong solution of another QOTE puzzle from all 504 QOTE puzzles.
The capability of the game to identify more accurate sets of infeasible test path candidates.	The number of remaining unsolved "MainQOTE" puzzles, compared to that of "MainRings".
Enjoyability of the games (concerning research Question 5).	The percentage of participants who selected QOTE as the answer to questions 1 to 5 of the questionnaire (Figure 11).
Better and more useful GWAP for test data generation.	Von Ahn and Dabbish [7] introduced three metrics: • Throughput = The mean number of problem instances (i.e., test paths) solved per human hour; • Average Lifetime Play (ALP) = The total amount of time an individual plays the game on average among all individuals who have played it; • Expected contribution = Throughput × ALP. If the expected contribution of QOTE is higher than that of Rings, we can conclude that QOTE is a more useful GWAP for test data generation than Rings.

Table 3. General comparison between the prior games and QOTE.

	Generated puzzles	Infeasible paths	Solved puzzles by wrong solutions of other puzzles
QOTE	504	232 out of 504 paths (part 1)/ 233 out of 504 paths (part 2)	151 paths (part 1)/ 151 paths (part 2)
Rings	194	82 out of 194 paths	0 path
Greenify	504	324 out of 504 paths	67 paths

Table 4. The time of playing the games on average in each session.

	Part 1 of the experiment	
	Number of solved puzzles by RingsFirst/time	**Number of solved puzzles by QOTEFirs/time**
MainRings	(I) 99/ $95' : 42''$	(II) 100/ $96' : 33''$
MainQOTE	(III) 134/ $45' : 38''$	(IV) 135/ $42' : 17''$
ExtraQOTE	(V) 136/ $46' : 12''$	(VI) 133/ $50' : 20''$
	Part 2 of the experiment	
	Number of solved puzzles by GreenifyFirst/time	**Number of solved puzzles by QOTEFirst/time**
MainGreenify	(VII) 177/ $155' : 31''$	(VIII) 180/ $148' : 32''$
MainQOTE	(IX) 271/ $75' : 49''$	(X) 266/ $78' : 28''$

Table 5. The result of the players' answers to the questionnaires.

Questions	**Mean rate to QOTE by the players in part 1 /part 2 of the experiment**	**Mean rate to Rings by the players in part 1 /mean rate to Greenify by the players in part 2**	**t-test parameters**		
			The NULL hypothesis in part 1 /null hypothesis in part 2	**Alternative hypothesis in part 1 /alternative hypothesis in part 2**	***p*-value for part 1 /*p*-value for part 2**
1- The game was engaging	4.3/4.4	2.80/3.2	Rings is more engaging/ Greenify is more engaging	QOTE is more engaging/ QOTE is more engaging	$(< 0.01)/$ (< 0.01)
2- The game was boring	1.83/2	2.76/2.5	QOTE is more boring/QOTE is more boring	Rings is more boring/ Greenify is more boring	$(< 0.01)/$ (< 0.01)
3- I would have liked to continue playing the game	3.96/4.1	2.56/3.2	In this case, Rings is better/In this case, Greenify is better	In this case, QOTE is better/In this case, QOTE is better	$(< 0.01)/$ (< 0.01)
4- While playing the game, I forgot where I was	3.86/3.92	2.63/3.1	In this case, Rings is better/In this case, Greenify is better	In this situation, QOTE is better/ In this case, QOTE is better	$(< 0.01)/$ (< 0.01)
5- The game was fun	4.03/4.21	2.5/2.91	Rings is more fun/ Greenify is more fun	QOTE is more fun/ QOTE is more fun	$(< 0.01)/$ (< 0.01)
6- The game's user interface was easy to use	3.5/3.6	2.66/3.2	Rings is easier to use/ Greenify is easier to use	QOTE is easier to use/ QOTE is easier to use	$(< 0.01)/$ (< 0.01)
7- Playing the game was hard	1.7/1.52	2.86/2.78	Playing QOTE is harder/Playing QOTE is harder	Playing Rings is harder/ Playing Greenify is harder	$(< 0.01)/$ (< 0.01)

Table 6. The results of GWAP metrics for all games.

		Throughput (problem instance per human-hour)	ALP (hour)	Expected contribution
Part 1	Rings	62.26	0.063	3.95
	QOTE	192.85	0.028	5.39
Part 2	Greenify	78.60	0.103	8.11
	QOTE	204.61	0.052	10.63

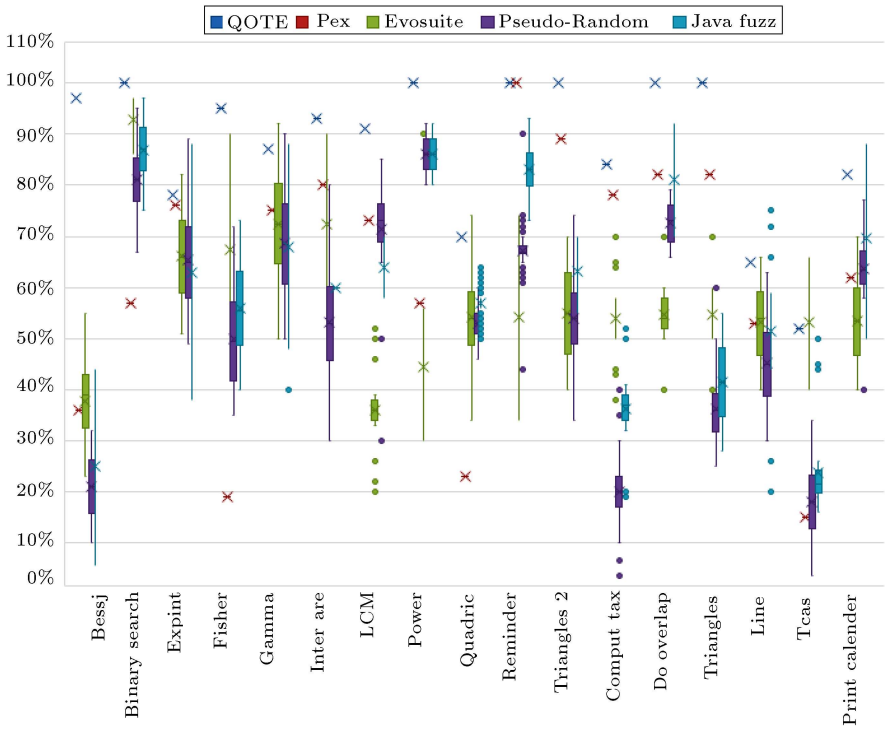


Figure 12. Calculated mutation scores by QOTE, Evosuite (a hybrid method of search-based and constraint-based testing), Pex (a symbolic execution method), Fuzz testing (a random method), and Pseudo-Random (a random method).

(Pex), 57% (Evosuite), 55% (Pseudo-Random), and 60% (Fuzz testing). All results are statistically significant (We performed four t-tests, all resulting in p-value < 0.01). See Figure 12 for detailed results.

There are reasons for this outperformance, which are explained in Section 5.3.1 in detail. Nevertheless, we intend to concentrate on one of these reasons here. Since QOTE aims to cover paths, it provides more test cases compared to the other tools based on weaker coverage criteria. To reduce the effect of the coverage criteria, we carried out another experiment.

In the new experiment, QOTE was again compared with one of the competitors based on the same coverage criterion. We selected Pex among the competitors because of its block coverage criterion, which is the weakest criterion among the criteria used by the competitors. It is worth mentioning that QOTE was not again implemented based on the block coverage criterion; instead, the data generated by QOTE was down-sampled.

Even after downsampling for the block coverage criterion, QOTE performs better regarding the average mutation score (75%) than Pex (62%). Results are statistically significant (We performed t-tests, which resulted in p-value< 0.01). See Figure 13 for details.

5. Discussion

In this section, we discuss the results presented in the previous section. To this end, QOTE's features are first described, and then we review the results concerning the research questions.

5.1. Solving the issues of Rings and Greenify via the features of QOTE

Two issues of Greenify and Rings are the scalability problem and the complexity of visualizing and handling the program's wide range of input parameters. Both

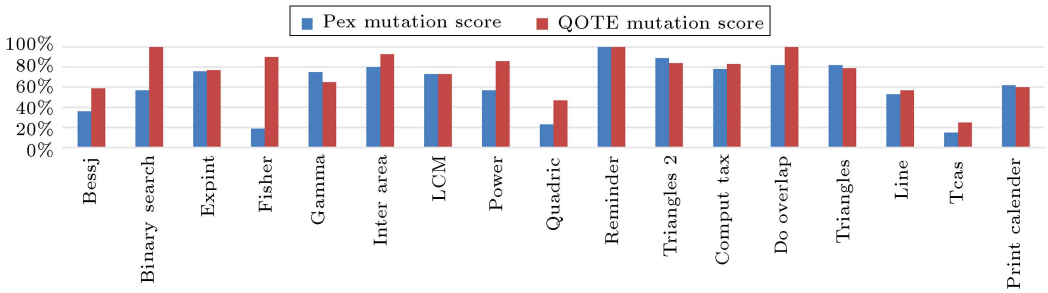


Figure 13. Calculated mutation scores by QOTE and Pex based on the block coverage criterion.

mentioned issues are resolved by QOTE, as described in Section 3.2.2.

The other issue with Greenify and Rings is that in each level, only one CFG path is shown to the players. The path may present unsolvable puzzles to the players. These puzzles lead to frustration for the players. QOTE shows all paths of the CFG at a given game level. At any level of QOTE, there is at least one solvable puzzle. In addition, the likelihood that a given level in QOTE and Greenify/Rings has at least one answer is 100% and 56%, respectively. In addition, QOTE stores players' right and wrong solutions. This may result in a smaller set of likely infeasible paths by QOTE.

Rings was unable to visually model nonlinear and more complex path constraints. Also, The Rings' puzzles are very long, affecting the puzzles' difficulty. In QOTE, the conditional statements are treated as black boxes. This means that in QOTE, the players only observe the results of evaluating the conditional statements based on the selected input values and are not involved in the internal structure of the conditional statements. In addition, Rings was designed on top of symbolic execution [9]. Subsequently, it suffers from every issue and limitation of symbolic execution (i.e., path explosion, external library calls, etc. [21]).

5.2. Reviewing the results concerning the research questions

In this section, we walk through the research questions and discuss the implications of the evaluation results concerning these questions.

Answer to Question 1: The time of playing the games, mentioned in Table 4, shows that the players generated test data faster when playing QOTE.

Answer to Question 2: According to the players' answers to Question 6 in their questionnaire, the QOTE's user interface was easier to use.

Answer to Question 3: The participants stated in their questionnaire that QOTE was less complicated than the prior games. In comparison to Rings, the Rings' game design was not able to generate puzzles for 310 out of 504 test paths due to their mathematical complexity. On the other hand, QOTE generated puzzles for all 504 test paths. This answers the second part of this research question and shows QOTE's applicability in test data generation for more complex programs.

Answer to Question 4: In QOTE and Greenify, 151 and 67 puzzles were respectively solved by wrong solutions of other puzzles before being solved by any player.

Answer to Question 5: The collected data from the questionnaires and QOTE's higher expected contribution, compared with the prior games, demonstrate that QOTE was more enjoyable.

Answer to Question 6: The results mentioned in Section 4.2 show that the data generated by QOTE found more faults than four other tools.

5.3. Comparison with automatic tools

5.3.1. The degree of effectiveness in detecting failures

We have used PIT to create 867 mutants for all benchmarks in Table 1. These mutants are divided into five groups based on the type of changes that they have made. As shown in Table 7, a few mutants of groups 2, 4, and 5 survived. Therefore, only groups 1 and 3 are studied as follows:

1. Random methods have the worst performance in killing "changed conditional boundary" mutants since merely data with specific values are required for killing these mutants. While human and intelligent approaches can easily generate these values.
2. Pex and Evosuite have the worst performance in killing "Replaced operator with another operator" mutants for the following reasons:
   a. Providing more data could be more effective in killing these mutants. Since, in comparison to QOTE, Pex, and Evosuite are based on weaker test coverage criteria, they could finish their work by generating less data. Therefore, QOTE outperforms its competitors in terms of killing mutants;
   b. The number of generated data is one of many factors affecting data quality. Therefore, it is

Table 7. The number of survived mutants by generated data using each method.

	Mutant	Random	QOTE	Pex	Evosuite
1	Changed conditional boundary (e.g., change $x > 10$ to $x >= 10$)	103	48	88	82
2	Negated conditional (e.g., change $x > 10$ to $x <= 10$)	21	6	31	21
3	Replaced operator with another operator (e.g., change $x + 10$ to $x - 10$)	120	21	141	161
4	Replaced return of value with new value (e.g., change return x to return $x + 1$)	3	1	6	1
5	Changed increment in a loop (e.g., change for $(;;i + 1)$ to for $(;;i - 1)$)	3	4	10	7

better to consider the variety of the generated data as a factor affecting the quality of data;

c. Mutants of group 3 are usually survived when we use Zero and One values. For example, the mutant that replaces $1 + x$ with $1 - x$ is not killed by $x = 0$. Studies on data generated by Pex and Evosuite reveal that these tools mostly generate Zero and One.

Generated data by (non-expert) players usually do not suffer from the above drawbacks and, thus, have high quality to kill the mutants.

5.3.2. The usefulness of GWAPs to apply players' intelligence

Players behave non-randomly during gameplay, and they act intelligently. A large number of successful GWAPs is hard evidence to show the players' brain does not function by mere random. QOTE, as a GWAP, applies players' intelligence as follows:

1. Since an individual learns and thinks differently from one another, the playing behavior differs from one player to another. For example, a player tries to set the locks to zero while the other one sets the lock from larger to smaller numbers. The advantage of the different characteristics of the players is that there are as many intelligent algorithms as the number of players to solve the puzzle;
2. Based on the difference between the players, the generated passcodes have a higher variety in values compared to the automated approaches;
3. Top game players are valuable assets for a GWAP. There might be only one top game player among a hundred. However, that top player may solve a complicated puzzle very efficiently, so his answer to the puzzles is invaluable for our purpose.

5.3.3. How and where the players use their intelligence

At the beginning of the gameplay and in the initial levels, players blindly search for passcodes; but during the game routine, the players learn how to find the passcodes through their intelligence. In the following, we mention opportunities for the players to analyze and learn, leading to an improvement in playing the game:

1. Usually, a series of conditional statements are repeated in most program codes. Hence, the corresponding game elements are repeated at different levels of the game;
2. The process of dialing the locks to turn gear trains and unlock safety boxes creates a visual perspective for the players. The players make better use of their intelligence from the visual perspective;
3. Players typically look for a cause-and-effect relationship between combination locks and gears. For this reason, humans do not get stuck in a repetitive gear train.

6. Conclusions

A significant task in the process of software testing is test data generation. Automatic test data generation methods have been extensively studied. But software industry still depends on human resources due to the low maturity level of automatic techniques. Therefore, finding new ways to improve human-based test data generation is still an important issue. For this purpose, we used the concept of Game With A Purpose (GWAP) to reduce the costs of human-based test data generation and increase its appeal to engage even non-technical people. The main idea behind our work is that GWAPs transform a normal and monotonous task into an exciting mission. In this way, numerous inexpensive players with no special technical ability become engaged with the test data generation activity through game playing.

Rings and Greenify were early works employing GWAP in human-based test data generation with promising results [9,10]. However, they had certain problems, including limitations on the number of program inputs, scalability, etc. With these issues in mind, we designed Quest Of Treasure Explorer (QOTE) to improve the applicability of GWAP to test data generation. The experimental results show that not only QOTE provides a more enjoyable experience for the players, but also it showed a better test data generation tool. In other words, using QOTE, more

test data was generated in a shorter time by more entertained players.

Regarding test data capabilities, we have conducted an experiment to compare the performance of QOTE versus Pex, Evosuite, Fuzz testing, and Pseudo-Random, which are well-known automatic approaches. The calculated mutation scores of each approach show that the generated test data by QOTE find more faults than the automatic approaches mentioned.

References

1. Ammann, P. and Offut, J., *Introduction to Software Testing*, In Cambridge University Press (2016).
2. Jain, N. and Porwal, R. "Automated test data generation applying heuristic approaches-a survey", In *Software Engineering, Springer, Singapore*, pp. 699–708 (2019).
3. Valueian, M., Attar, N., Haghighi, H., et al. "Constructing automated test Oracle for low observable software", *Scientia Iranica*, **27**(3), pp. 1333–1351 (2020).
4. Ouni, A. "Search based software engineering: challenges, opportunities and recent applications." In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pp. 1114–1146 (2020).
5. Kirkpatrick, G. "Welcoming All Gods and Embracing All Places": Computer Games As Constitutively Transcendent of the Local.", In *Game History and the Local. Palgrave Macmillan, Cham*, pp. 199–219 (2021).
6. Zachos, G., Paraskevopoulou-Kollia, E.A., and Anagnostopoulos, I. "Social media use in higher education: A review", *Education Sciences*, **8**(4), p. 194 (2018).
7. Von Ahn, L. and Dabbish, L. "Designing games with a purpose", *Communications of the ACM*, **51**(8), pp. 58–67 (2008).
8. Siu, K.A. "Design and Evaluation of Intelligent Reward Structures in Human Computation Games", PhD Thesis, Georgia Institute of Technology (2021).
9. Amiri-Chimeh, S., Haghighi, H., Vahidi-Asl, M., et al. "Rings: A game with a purpose for test data generation", *Interacting with Computers*, **30**(1), pp. 1–30 (2017).
10. Moosavi, Sh., Haghighi, H., Sahabi, H., et al. "Greenify: A game with the purpose of test data generation for unit testing", In *Fundamentals of Software Engineering: 8th International Conference*, FSEN 2019, Tehran, Iran, Springer International Publishing, pp. 77–92 (2019).
11. Gaurav, D., Kaushik, Y., Supraja, S., et al. "Empirical study of adaptive serious games in enhancing learning outcome", *International Journal of Serious Games*, **9**(2), pp. 27–42 (2022).
12. Tillmann, N., De Halleux, J., Xie, T., et al. "Pex4fun: Teaching and learning computer science via social gaming". In *2012 IEEE 25th Conference on Software Engineering Education and Training*, pp. 90–91 (2012).
13. Tillmann, N., Bishop, J., Horspool, N., et al. "Code hunt: Searching for secret code for fun", *7th International Workshop on SearchBased Software Testing*, pp. 23–26 (2014).
14. Fava, D., Shapiro, D., Osborn, J., et al. "Crowdsourcing program preconditions via a classification game", *38th International Conference on Software Engineering*, pp. 1086–1096 (2016).
15. Rojas, J.M. and Fraser, G. "Code defenders: A mutation testing game", *2016 IEEE Ninth International Conference on Software Testing*, Verification and Validation Workshops (ICSTW), pp. 162–167 (2015).
16. Chen, N. and Kim, S. "Puzzle-based automatic testing: Bringing humans into the loop by solving puzzles", *27th IEEE/ACM International Conference on Automated Software Engineering*, IEEE, pp. 140–149 (2012).
17. Michael, C.C., McGraw, G., and Schatz, M.A. "Generating software test data by evolution", *IEEE Transactions on Software Engineering*, **27**(12), pp. 1085–1110 (2001).
18. Walkinshaw, N. and Fraser, G. "Uncertainty-driven black-box test data generation", *2017 IEEE International Conference on Software Testing*, Verification and Validation (ICST), pp. 253–263 (2017).
19. Papadakis, M., Kintis, M., Zhang, J., et al. "Mutation testing advances: an analysis and survey", *Advances in Computers*, Elsevier, **112**, pp. 275–378 (2019).
20. Coles, H., Laurent, T., Henard, C., et al. "Pit: a practical mutation testing tool for java", *25th International Symposium on Software Testing and Analysis*, ACM, pp. 449–452 (2016).
21. Malte, M. and Howar, F. "An ensemble of tools for dynamic symbolic execution on the Java Virtual Machine (competition contribution)", In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference*, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, Cham: Springer International Publishing (2022).

Biographies

Sharmin Moosavi is a PhD student in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. She received her BSc and MSc from the software engineering Group of the Department of Computer Engineering, Isfahan University, Isfahan, Iran. In addition, she is a faculty member of Islamic Azad University. Her research interests include serious games, computation games, software testing, and debugging.

Mojtaba Vahidi Asl is an Assistant Professor in the Computer Science and Engineering faculty at Shahid Beheshti University. He received his BSc from the Amirkabir University of Technology in 2005 and his

MSc and PhD from the Iran University of Science and Technology in 2008 and 2014, respectively. His research interests include Software testing, Fault localization, Program repair, and Computer games.

Hassan Haghighi received his PhD degree in Computer Engineering-software from Sharif University of Technology in 2009 and is currently a Professor in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. His main research interests include formal methods, software testing, and data quality.