3C3-IOS-2-1

Intelligent Level Generation for Super Mario using Interactive Evolutionary Computation

Ching-Ying Cheng Ya-Hung Chen Tsung-Che Chiang

Department of Computer Science and Information Engineering,
National Taiwan Normal University, Taiwan

The computer game industry is growing rapidly with the yearly revenue rising up to tens of billion dollars. The market is huge but meanwhile very competitive. A computer game with static and deterministic design will be discarded after a short playing duration. Thus, we need a game that can keep creating interesting and challenging contents by adapting to customers' preferences. In this research we aim to design an intelligent level generator for the famous Super Mario game. The generator collects the player's data in a base game level and identifies his/her skill and playing tendencies such as jumping and coin-collecting. Then, it generates a customized game level by fitting to these tendencies. The level generator is directly evaluated by human players, and the parameters of the level generator are optimized through interactive evolutionary computation. A simple regression model is built and applied to make the evolutionary process more efficient.

1. Introduction

The computer game industry is growing rapidly with the yearly revenue rising up to tens of billion dollars. The market is huge but meanwhile very competitive. Players have a lot of choices of platforms from early Nintendo NES to the PlayStation 3, Wii, Xbox 360 and personal computers (PC). The booming of PC games, especially the social web games or online games, increases the pressure to the game industry. Every week a variety of games are available and compete to enter the market, and thus how to extend the life cycle of game products has become an important topic.

Games with static and deterministic design will be discarded after a short playing duration. Difficulty is also an important factor affecting players' interests in the game. If the game is too difficult, players may lose confidence and interest in the game; if the game is too simple with little challenge, it also reduces the game's attraction. Thus, we need a game that can keep creating interesting and challenging contents by adapting to customers' preferences. In this research we aim to design an intelligent level generator for the famous Super Mario game. The research is motivated by the 2012 Mario AI Championship [Shaker 2010, Shaker 2012].

Super Mario is an action game with hills, obstacles, enemies, blocks, coins, and gaps constituting the scenes. We design a game level generator to furnish those objects to create levels. The game generator collects the player's data in a base game level and identifies his/her skill and playing tendencies such as jumping and coin-collecting. Then, it generates a customized game level by fitting to these tendencies. The level generator is directly evaluated by human players, and the parameters of the level generator are optimized through interactive evolutionary computation. A simple regression model is used to improve the efficiency of the evolution.

The rest of this paper is organized as follows. Section 2 describes the system architecture, player skill and tendencies, and specialized game zones and models in the level generator. Section 3 presents how we use the interactive evolutionary optimization technique to improve the parameter setting of the proposed level generator. Experiments and results are given in Section 4. Section 5 provides conclusions and future directions.

2. System Architecture

2.1 Flow of the system

A level generator creates two game levels in a round. The first level is called the Test Level, and the second one is called the Customized Level. First, a tester (i.e. a human player) plays the Test Level, during which the generator records the playing information (game play) of this tester, such as the times of death, places of death, level clear time, the number of collected coins, etc., into the game play database. Then, the generator analyzes the information and determines the skill and tendencies of this tester. The Customized Level is created by combining different game zones with different appearing rates based on the tester's skill and tendencies.

There are several versions of level generators in the system, and they are evaluated by human players. After playing both levels, the system will ask the tester to feedback a score of the Customized Level to indicate how much he/she likes the level. The score is stored in the database and will be utilized as a fitness of this version of level generator in the evolutionary algorithm. Once all candidate versions of the level generator (with different parameter settings) are tested/played by enough times, the evolutionary algorithm will select the better versions to produce new versions to replace worse ones. The survived old versions and new versions of level generators will again be evaluated by human players. Repeating this interactive optimization process, we can find a level generator with good parameter setting.

Contact: Tsung-Che Chiang, Department of Computer Science and Information Engineering, National Taiwan Normal University, tcchiang@ieee.org

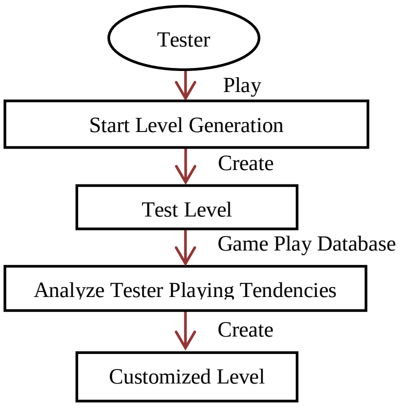


Fig. 1 The game level generation process

2.2 Player skill and tendencies

After a tester finishes playing the Test Level (whether he/she clears the level or not), the generator records the playing information (game play) listed in Table 1.

Table 1 Game Play Information

Data items
Number of lives
Level clear time
Times of death
Times killed by enemies
Times of death by falling into gaps
Number of passed gaps
Times of jumping
Times of redundant jumping
Times of sprints
Number of kills
Number of killed enemies of each type
Number of appearing enemies
Number of appearing enemies of each type
Number of collected coins
Times of hitting blocks
Times of upgrades

We use the game play information to determine this tester's skill and playing tendencies. Hereafter we call them player attributes. Each attribute has five levels, 1 to 5. A higher value means higher skill or tendency. The four tendencies are jumping, coin collecting, enemy killing, and block destroying.

(1) Skill

We evaluate the player's skill (*S*) primarily based on the level clear time and times of death. Shorter clear time and fewer times of death imply that the player has better skill, and we assign a larger value of *S*. We find that the clear time (the time required to clear the level or to use up all lives of Mario) of the Test Level is between 50 and 130 seconds. When the player cannot clear the level, we set *S* by 1; when the player clears the level within 50 seconds, we set *S* by 5. Levels of 2 to 4 are set according to the clear time and the times of death. Our level generator will create difficult levels (with more enemies and gaps) for players with high skills.

For example, in the Customized Level the number of red turtles increases linearly as the *S* value increases from 1 to 5. Hedgehog (one kind of strong enemy in the game) appears only when the player's skill is greater than 2, and flying enemies appear only when the skill is greater than 3.

(2) Jumping

The jumping tendency (*J*) is determined by the times of jumps a player does. We classify jumps into intentional jumps and unintentional jumps. Jumps that hit blocks, collect coins, or hit enemies, are regarded as intentional. We calculate a rate R_{JUMP} by the ratio of the number of unintentional jumps to the total number of jumps. We assign *J* by 1 when R_{JUMP} is less than 0.4, and increase *J* as R_{JUMP} increases.

(3) Coin collecting

Coin collecting tendency (*C*) is determined by the ratio of the number of collected coins to the total number of coins. The coins may be put in the scene explicitly or be hidden in the blocks. Let *x* and *y* denote the number of collected coins in the scene and in the blocks, respectively; let *X* and *Y* denote the total number of coins in the scene and in the blocks, respectively. We calculate a rate R_{COIN} by $(0.4x + 0.6y)/(X+Y)$. We assign a larger weight to the number of coins collected from the blocks since we think players hitting blocks to get coins have higher coin-collecting tendency. We assign *C* by 1 when R_{COIN} is less than 0.2 and increase *C* as R_{COIN} increases.

(4) Enemy killing

Enemy killing tendency (*E*) is determined based on the ratio of the number of enemies killed to the total number of enemies. As we put different weights on coins collected from different places, we put different weights on enemies killed by different ways. We put a lower weight on the number of enemies killed by turtle shells and a higher weight on the number of enemies killed by fire balls.

(5) Block destroying

Block destroying (*B*) is determined based on the ratio of the number of destroyed blocks to the total number of blocks. Again, we put different weights on the destroyed blocks of different types. A higher weight is put on the empty blocks, and a lower weight is put on the blocks with items inside.

2.3 Specialized game zones and models

Based on the five identified player attributes (skill and four tendencies), we develop five categories of game zones specifically. Each category contains of two or three zones, as shown in Fig. 2.

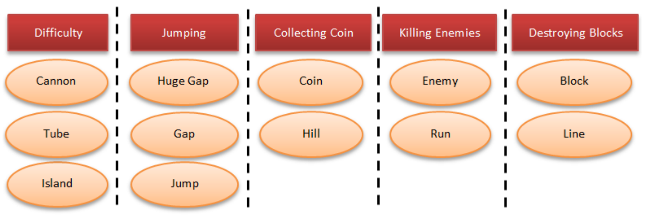
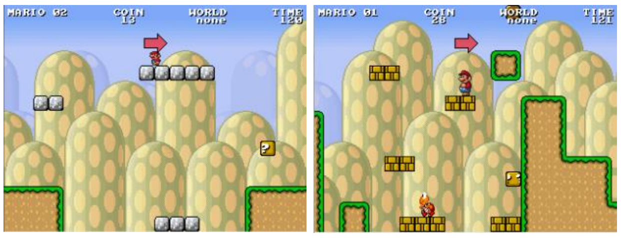


Fig. 2 Twelve specialized zones in five categories

The level generator creates the Customized Level by selecting and combining different game zones. Zones are selected in a probability proportional to the corresponding attribute value. Assume that the values of the five attributes (skill, jumping, coin collecting, enemy killing, and block destroying) of a player are 1, 2, 3, 4, and 5. Then, the zones of the "jumping" category will be selected with probability as (3×2)/(3×1+3×2+2×3+2×4+2×5).

Each kind of zone corresponds to a main attribute. For example, the Huge Gap zone is developed for players with high jumping tendency. Each zone is composed of multiple models to increase the variety of scenes and to fit a secondary attribute. For example, the Huge Gap zone consists of two models to further fit the players' skill: a simple model without enemy for players with lower skill (Fig. 3(a)) and a hard model with enemies for players with higher skill (Fig. 3(b)). In total we have 28 kinds of models.



(a) (b)

Fig. 3 Huge Gap Zone: (a) easy model (b) difficult model

In addition to the above-mentioned specialized zones, we also develop a boss stage to put at the end of the Customized Level for more fun. The difficulty of this stage depends on the player's skill and enemy-killing tendency.

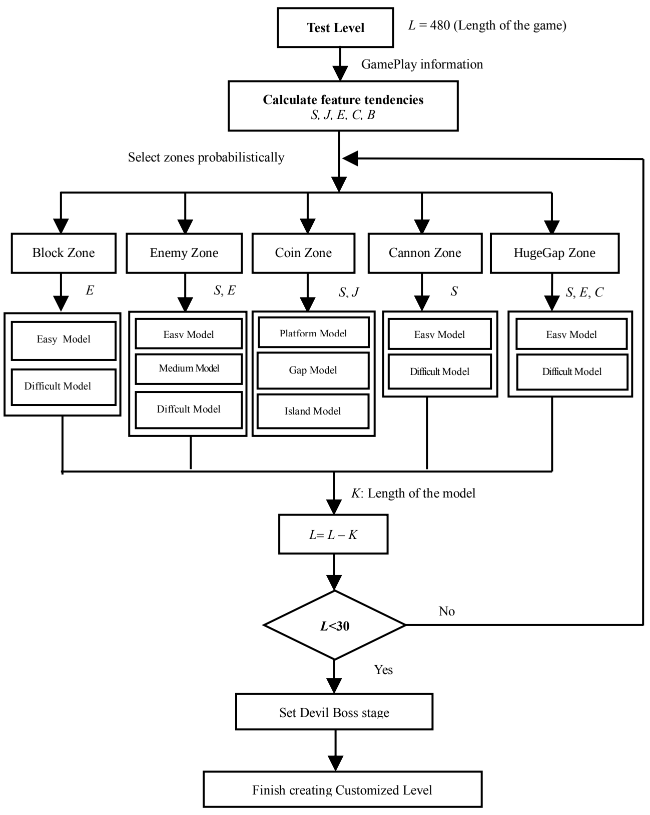
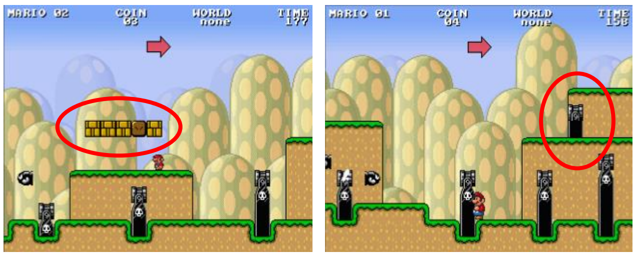


Fig. 4 Creation of the Customized Level

Fig. 4 is the process of creating the Customized Level. After the player finishes the Test Level and his/her attributes are decided, the level generator creates the Customized Level by concatenating different zones/models accordingly. It selects a zone based on the main attribute and then selects a model based on the secondary attribute. The length of a level is 480 units (each of 30 pixels). A model usually has the length between 20 and 25 units. Thus, a level is composed of about 20 models and a final boss stage. Due to the space of limitation, in Fig. 4 we only present one zone and its models for each attribute category. These five zones are representative one for each player attribute, respectively. We will show them with brief descriptions.

(1) Cannon zone

There is no cannon in the Test Level, and thus players would have fun and surprise to see it in the Customized Level. The Cannon zone has two models, the easy and difficult models (Fig. 5). In the easy model cannons are located at the same plane, whereas in the difficult model cannons could be located at different planes so that players have to worry about bullets coming at different heights. The number, position, and height of the cannons are generated randomly to make different scenes.



(a) (b)

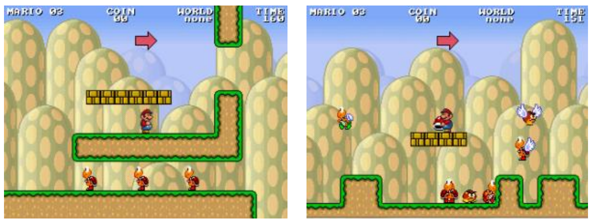
Fig. 5 Cannon Zone: (a) easy model (b) difficult model

(2) Huge gap zone

There is a huge gap in this zone. Players cannot pass the zone simply by a single jump. They have to jump over blocks. This zone is developed for players who love jumping. It consists of the easy and difficult models. The easy one contains no enemy and is selected in higher probability when the player has lower skill. The difficult one contains enemies and is chosen more likely when the player has higher skill or enemy-killing tendency.

(3) Enemy zone

The enemy zone consists of three models: easy, medium, and difficult. The models are selected according to the player's skill. In the easy model there is a way to pass without encountering any enemy, but in the difficult model the player has to kill the enemy to pass through (Fig. 6).

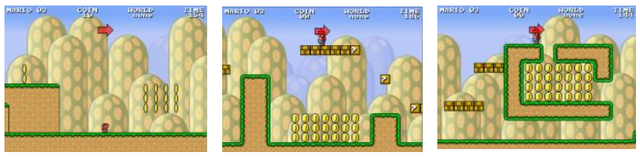


(a) (b)

Fig. 6 Enemy Zone: (a) easy model (b) difficult model

(4) Coin zone

The coin zone is for players love collecting coins. It has three models: platform, gap, and island (Fig. 7). The multiple models are created for the variety of scenes. The gap and island models appear more frequently for players with higher skill and jumping tendency.

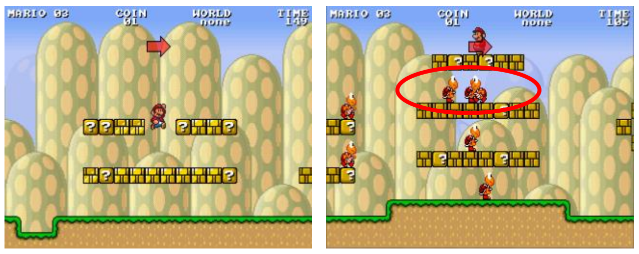


(a) (b) (c)

Fig. 7 Coin Zone: (a) platform (b) gap (c) island models

(5) Block zone

The block zone separates into the easy and difficult models, one without enemy and the other with enemies. Roughly, the ratios of empty blocks, coin blocks, and item blocks are 50%, 40%, and 10%, respectively. The level generator will increase the ratio of coin blocks for players with higher coin-collecting tendency and decrease the ratio of item blocks for players with higher skill. There are about 5% hidden blocks with coins inside. These blocks will appear more frequently when the player has higher coin-collecting tendency and longer clear time.



(a) (b)

Fig. 8 Block Zone: (a) easy model (b) difficult model

3. Optimization of the Level Generator by Interactive Evolutionary Computation

3.1 Evolutionary algorithm (EA)

EAs [Goldberg 1989, De Jong 2002] are algorithms that attempt to solve complex problems by mimicking the evolutionary process in the nature. It starts with an initial population of individuals, each of which is an encoded form of a solution to the target problem. Each individual is given a fitness value to reflect the quality of the corresponding solution. Individuals with better solution quality are selected as parents in higher probability. Then, the parents produce the offspring (new solutions) through genetic operators including crossover and mutation. Among the old population and new offspring, good individuals are selected to survive to be the population in the next generation. The above steps repeat generation by generation to evolve the individuals toward the (near) optimal solutions. EAs have been applied to game AI design such as Pac-man controller [Gallagher 2003], racing car controller [Togelius 2005] and game parameter optimization such as car setup [Muñoz 2010], tower defense [Avery 2011], and wizard skill balancing [Wong 2012].

3.2 Interactive evolutionary optimization

We identify the player's five attributes and develop game zones particularly fitting to these attributes. Players with higher jumping tendency will encounter more jumping zones/models in the created Customized Level. What we need now is a base appearing ratio of the five kinds of scenes that meets players' general expectation. Based on this base ratio, we then adjust the ratio of zones according to each player's specific tendency. Let p_i ($i = 1, 2, \ldots, 5$) denote the base ratio of the zones of a category of attribute i, n_i denote the number of zones in the category, and a_i denote the attribute value of a player. The probability of selecting a zone from a category of attribute i is now $Prob(i) = p_i n_i a_i \Big/ \sum_{i=1}^{5} p_i n_i a_i$. (In the example of Section 2.3, we assume $p_i = 1$ for all attributes.)

Now, tuning the five parameter values p_i of the level generator is to be solved as an optimization problem with five discrete decision variables. We will use the evolutionary algorithm to solve this problem. The chromosome representation is a five tuple of integers. To reduce the solution space, we restrict the values of each variable to only three levels: low (1), medium (5), and high (10). An individual (i.e. a set of parameter values) corresponds to a version of the level generator. The quality of each version of level generator is evaluated by human players' feedback. By incorporating human into the evaluation procedure of the evolutionary algorithm, we come up with an interactive evolutionary computation-based optimization approach.

To achieve the interaction, we set up a server running a database and an evolutionary algorithm. The database stores the current population of individuals (the parameters of level generators), their times of being played, and fitness values. We also modify the program [Shaker 2012] of a Java version of the Super Mario game to put our level generator inside and add communication procedures. After a player finishes playing the Test Level, the program will connect to our server to get one set of parameters (p_i). Then, our level generator will create a Customized Level based on the parameter values and player attributes (a_i). When the player finishes playing the Customized Level, the program will ask him/her to return a score to the server to be stored.

(1) Initialization

We generate the initial population by random assignments. Each individual is composed of five integers, whose values are 1, 5, or 10 randomly.

(2) Evaluation

To help the evolutionary process, human players download the (modified) Super Mario program from our web site. When the program connects to our server to retrieve an individual, our server will give it the one with the minimal times of plays. At the end of the game, the program sends back a score (ranging from 1 to 5), which will be stored with the played individual.

To avoid the bias caused by too few feedbacks, each individual needs to be played for K times (i.e. K scores are received). Initially the value of K is set by 5. After each generation, its value increases by one. Fig. 9 shows the flow of the evaluation procedure.

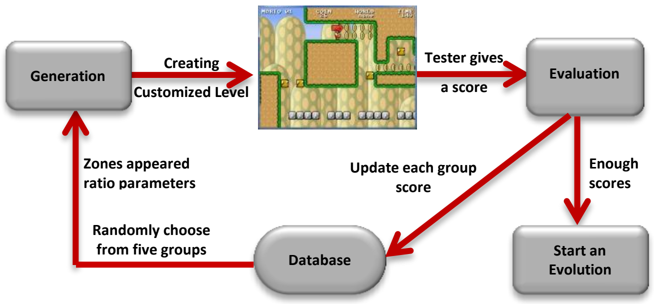


Fig. 9 The flow of the evaluation procedure

(3) Reproduction

Once all individuals in the current population have been evaluated for K times, we select the two best individuals as parents. Each of them undergoes the mutation to produce a new offspring. In the mutation operation, two among the five parameters are selected and assigned a random value. The remaining three parameters take the original values from the parent. The two offspring replace the worst two individuals in the current population. The new generation begins with K increasing by one. It means that the three old individuals must be evaluated by human players for one more time, and the two new individuals must be evaluated by K times.

(4) Stopping criterion

We do not set an explicit stopping criterion. In fact, as long as the feedbacks from players come, the evolutionary process can keep going. In other words, how far the evolutionary process goes depends on how many feedbacks we get from human players. In our experiments, we encounter the difficulty of finding a large number of players. Since each feedback is valuable, we do not want to waste it on useless individuals. This leads us to estimate the fitness of individuals through an approximation method. Here we resort to the regression model.

3.3 Regression model

As mentioned, once all individuals in the current population are evaluated for enough times, two of them are used to generate two offspring. These two offspring will enter the population and consume $2 \cdot K$ evaluations from human players in the new generation. If they are bad individuals, the evaluations are wasted. Thus, we hope to find “potential” individuals to be evaluated by players. We build a regression model based on historical data and then use it to calculate the approximated fitness of the produced offspring.

The input of the regression model is the individual, that is, five player attributes. The output is the expected score from the player. We assume the five attributes are independent from one another. The model for the approximated score is defined by

$$f(a_1, a_2, a_3, a_4, a_5) = c_1 + c_2 \times a_1^2 + c_3 \times a_1 + c_4 \times a_2^2 + c_5 \times a_2 + c_6 \times a_3^2 + c_7 \times a_3 + c_8 \times a_4^2 + c_9 \times a_4 + c_{10} \times a_5^2 + c_{11} \times a_5 \quad (1)$$

Each pair of an individual and one feedback score serves as a training data point. Given n historical data points collected from previous generations, we try to fit the regression model (i.e. to find the values of the coefficients c_i, i = 1, 2, …, 11) to these points by minimizing the sum of square error.

With the regression model, we can pre-select the potential offspring for doing human evaluation. Since we have reduced the size of solution space to 3^5, which is not a very large number, we calculate the approximated fitness of all solutions. The best two solutions with the highest approximated fitness then replace the worst two individuals in the current population. On one hand, the regression model helps us to filter out bad individuals and improve the effectiveness of the evolutionary process. On the other hand, these new individuals still receive human evaluations, which provide true feedbacks to improve approximation accuracy.

4. Experiments and Results

4.1 Experimental setting

The level generator was implemented in Java programming language. The program was based on the code provided by Mario AI Championship 2012. The population size of the evolutionary algorithm was five. We collected 270 feedbacks, and the algorithm evolved for 10 generations.

4.2 Improvement by interactive evolutionary computation

The positive effect of the interactive evolutionary computation is shown in Fig. 10. The average score of individuals (versions of the level generator) is below 3 at the first generation and increases to 3.5 after 10 generations of evolution. It means that the final evolved level generators fit players better than the starting ones.

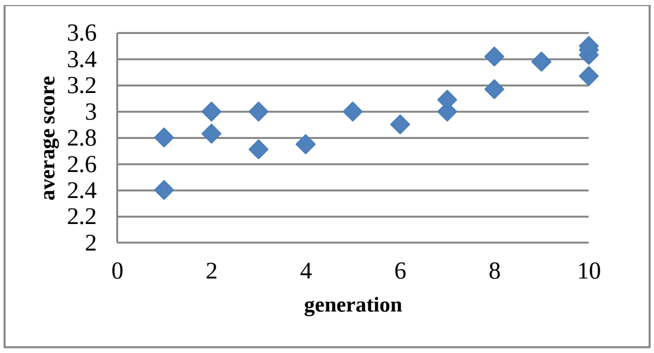


Fig. 10 average scores of individuals over generations

Table 2 shows the best three individuals found during the evolutionary process. The gene values correspond to the ratio of Block, Enemy, Coin, Cannon, and Huge Gap zones, respectively. In general, we can find that players (at least the tens of players who participate in our experiments) favor Cannon and Coin zones. Block and Huge Gaps zones should appear moderately, and Enemy zone should appear little.

Table 2 Best three individuals and corresponding appearing rate of zones

Top three sets	P_B	P_E	P_{Co}	P_{Ca}	P_H
(5,1,10,5,5)	0.192	0.038	0.385	0.192	0.192
(5,1,1,10,1)	0.278	0.056	0.056	0.556	0.056
(5,1,10,10,1)	0.185	0.037	0.37	0.37	0.037

4.3 Analysis of the regression model

Based on the collected user feedbacks, we build the regression model as follows:

$$s = f(a_1, a_2, a_3, a_4, a_5) = 4.048 + 0.026\times a_1^2 + (-0.34)\times a_1 + 0.007\times a_2^2 + (-0.114)\times a_2 + 0.023\times a_3^2 + (-0.358)\times a_3 + (-0.019)\times a_4^2 + 0.153\times a_4 + (-0.035)\times a_5^2 + 0.393\times a_5 \quad (2)$$

Fig. 11 shows the relationship between the expected average score and the actual average score. As we can see, there is still much room for improving the model accuracy. The following are some reasons for the inaccuracy.

(1) There may not be a single model suitable for all players. A level generator receiving high scores from a group of players may get low scores from another group.

(2) In our current model we ignore the interaction between player's attributes, but the interaction may exist.

(3) The number of human evaluation for the level generators is not enough. Some extreme (very high or very low) scores may cause unnecessary but significant impact.

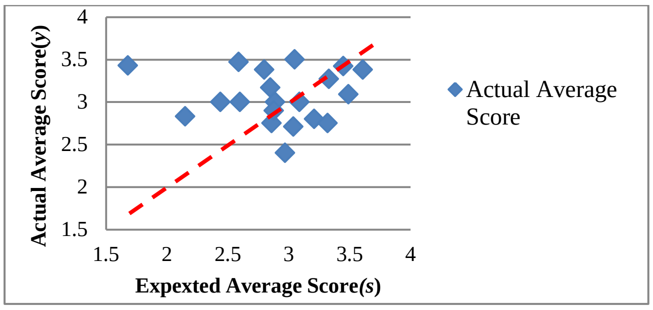


Fig. 11 Relationship between average scores and expected average score

5. Conclusions and Future Work

In this study we propose an intelligent level generator for the Super Mario game. The level generator collects game play information from human players, identifies their skill and playing tendencies, and generates customized levels accordingly. In the customized level, specific zones are selected with different rates, and the rates are determined by general user preference and target user preference. The general user preference is obtained by optimizing users' feedback scores through interactive evolutionary computation, and the target user preference is obtained by analyzing the single user's game play. To save the cost and time required by human evaluation, we build a regression model based on past data and use it to select potential level generators. Although the regression model is not very accurate, the level generator still fits players' tendencies better and better. Our level generator was also the winner of the Level Generation Track of the Mario AI Competition held in IEEE Conference on Computational Intelligence and Games (CIG) in 2012.

In this research the most difficult part may be the collection of enough human feedbacks for evolution. We try to solve the problem by predicting the feedback based on a regression model, but the accuracy needs much improvement. We could extend this work in the following directions: First, we need to collect data more carefully. On one hand, when a level generator receives a score deviating largely from the previous ones, we should consider disregarding this score. On the other hand, when a level generator keeps receiving close scores, we may consider not evaluating it any more. Second, we need to build the approximation model more accurately. A simple way is to take into account the interaction effect between player attributes in building the regression model. We can also resort to the literature about expensive optimization [Tenne 2010]. Finally, we need experiments to compare the interactive evolutionary optimization with and without the approximation model to know its effect.

Acknowledgment

We thank Dr. Shaker, Dr. Togelius, and Dr. Yannakakis for organizing the Level Generation Track of the Mario AI Competition. We also thank the players participated in our experiments. This research is supported by the National Science Council of Republic of China under research grant No. 101-2815-C-003-045-E.

References

[Avery 2011] P. Avery, J. Togelius, E. Alistar, and R. P. van Leeuwen, "Computational intelligence and tower defense games," *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 1084 – 1091, 2011.

[De Jong 2002] K. A. De Jong, *Evolutionary Computation*, the MIT Press, 2002.

[Gallagher 2003] M. Gallagher and A. Ryan, "Learning to play Pac-man: an evolutionary, rule-based approach," *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 242 – 2469, 2003.

[Goldberg 1989] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[Muñoz 2010] J. Muñoz, G. Gutierrez, and A. Sanchis, "Multi-objective evolution for car setup optimization," 2010 UK Workshop on Computational Intelligence, 2010.

[Shaker 2010] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N, Soreson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith, and R. Baumgarten, "The 2010 Mario AI championship: level generation track," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 4, pp. 332 – 347, 2010.

[Shaker 2012] N. Shaker, J. Togelius, and G. Yannakakis, Level generation track in Mario AI championship 2012, http://www.marioai.org/LevelGeneration/

[Tenne 2010] Y. Tenne and C. K. Goh, *Computational Intelligence in Expensive Optimization Problems*, Springer, 2010.

[Togelius 2005] J. Togelius and S. M. Lucas, "Evolving controllers for simulated car racing," *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 1906 – 1913, 2005.

[Wong 2012] S. K. Wong and S. W. Fang, "A study on genetic algorithm and neural network for mini-games," *Journal of Information Science and Engineering*, vol. 28, pp. 145 – 159, 2012.