Automatic Level Difficulty Adjustment in Platform Games Using Genetic Algorithm Based Methodology

Nirach Watcharasatharpornpong
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Payathai Road, Patumwan
Bangkok 10330, Thailand
Tel. +66817536782
notepe@gmail.com

Vishnu Kotrajaras
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Payathai Road, Patumwan
Bangkok 10330, Thailand
Tel. +66890212323
vishnu@cp.eng.chula.ac.th, ajarntoe@gmail.com

Abstract

In platform games, enemy behavior is not complicated. Therefore, challenges in such games come from the right mixture between enemies and environments of each level. Platform games require manual testing for tuning the game balance for mass audience. This is very time consuming. In addition, the difficulty of each level obtained is not guaranteed to suit individuals. Very few researches tackle how balanced levels can be generated automatically for individuals. This paper proposes a new methodology for using artificial intelligence to adjust games difficulty to suit players by automatically generating levels in platform games. The method is inspired by genetic algorithm. It is much easier to implement compared to an existing reinforcement learning based method, while still maintains similar gameplay quality. The new methodology also consumes less memory.

Keywords

Level design, platform game, genetic algorithm, automatic difficulty adjustment

1. Introduction

There are many researches that tackle the issue of difficulty adjustments in games. Most of them concentrate on enemy behavior adjustment. However, for platform games, their challenges come from learning to overcome obstacles presented by fixed enemies and game environments. Tuning the difficulty of platform games by adjusting the behavior of enemies will simply destroy the mechanic of such games.

An alternative approach for difficulty adjustment is to fine-tune the layout of game stages (including the placement of fixed behavior enemies). Kamnerdnond and Kotrajaras [1] proposed a model for automatically generating game environments according to players' performances for platform games. The model combined reinforcement learning with design methodologies. Their model, however, required a lot of memory storage because data for individual challenge plays and vote records from all previous challenges were needed during play.

In this paper, we present an alternative model for platform games level generation according to players' performances. We abandoned reinforcement learning approach and opted for an approach inspired by genetic algorithm. This approach resulted in less memory usage while still allowed levels to be produced effectively. Our prototype was made after Super Mario. We believed that by utilizing a well-known game mechanism, we would be able to demonstrate our model more clearly.

2. Related Works

Artificial intelligence applications for computer games can be grouped into two categories. The first category strives for the best possible agent behaviour. Genetic algorithm [2] and reinforcement learning [3] are prime examples of applications in this category. Bakkes et al. [4] created a team based AI for Quake III by using genetic algorithm to learn state-specific behavior for the team. Cole et al. [5] used genetic algorithm to evolve sets of parameters for bots in Counter Strike. Genetic algorithm was able to tune parameters as good as a highly experienced player could do in fifty generations, which was a relatively short time for training bots offline. Graepel et al. [6] used reinforcement learning to tune a fighting game AI character. Spronck et al. [7] introduced dynamic scripting, a form of reinforcement learning that could adjust an AI to win against its opponent in a relatively short time. The second category of artificial intelligence applications aims to adapt agents to suit players. Spronck et al. [7] demonstrated that dynamic scripting could be enhanced so that the game AI was able to scale its difficulty level to match its human opponent. Andrade et al. [8] applied reinforcement learning to match players’ performances with those of agents.

All these works concentrated on changing characters or agents’ behavior. For platform games, making an enemy character adapt or learn new behavior is not quite appropriate because the difficulty of platform games comes mainly from game environments and obstacles, not from enemy characters alone. Therefore the adjustment should be applied to the game environments instead. Pagulayan et al. [9] proposed a method for designing game environments to suit players. Challenges were put into each game level according to their difficulty. The aim was to teach collections of skills to players gradually. Players would then be able to improve their skills in order to tackle more difficult challenges and defeat game bosses. Björk and Holopainen [10] proposed that a game should have mechanisms for smoothing players’ learning curve in order to provide players with enough skills to progress while preventing boredom. However, these works mainly discussed good practice for manually designing game levels.

For automatic level generation of platform games, Kamnerdnond and Kotrajaras [1] used design methodologies from [9] and [10] together with reinforcement learning to create each suitable level for players to overcome. A level was formed from several challenges. Each challenge consisted of a sequence of continuous actions (players could not take a break while performing such action sequence). Each possible action was derived from players’ control skills. While playing a generated game level, a player’s performance was recorded. After the player finished each game level, the collected data was used as feedback to calculate the probability for each challenge to emerge in the next level. Initially, the system generated all possible challenges that did not contain more than a certain number of actions. Each challenge had its own difficulty score. All challenges were then divided into groups. Within each group, challenges were sorted by their difficulty score from low to high. After a player finished playing a game level, the number of successful and unsuccessful plays for each challenge was given to the reinforcement learning mechanism. A voting system was used. Each challenge could cast votes for more difficult or less difficult challenges of the same group. The spread of the voting range was determined by the play data. The probability for a challenge to be selected for the next level increased according to its voting score and decreased according to the number of times the player cleared that challenge (to prevent the challenges from being selected too often). For each challenge group, a certain number of challenges were chosen this way to construct the next level. Their system gave good experimental results. However, it consumed a lot of memory. Our paper presents an alternative level generation algorithm according to players' performance, with less memory usage.

3. Our Approach

We utilized a crossover-like mechanism to create new challenges in the next level, keeping them similar to previous challenges. A challenge was created based on skills we wanted players to learn. The graph in the middle of figure 1 shows all possible continuous action sequences in a Super Mario-like game. Action F represents the skill of throwing a fireball. Action M represents running, while E, J, and A represent avoiding enemies, jumping, and stomping on an enemy respectively. Each action can have varying difficulty scores depending on its target. For example, jumping to a small platform has higher scores than jumping to a large platform. A challenge is a sequence of these actions, as shown on the right of figure 1. In our approach, similar challenges were grouped. The difficulty of each challenge could be calculated from equation (1).

$$h_{dif} = \sum_{a_i \in \mathbf{A}} \left\{ [d_p(a_i) + \sum_{j \in \mathbf{E}} d_e(a_{ij}) + \sum_{k \in \mathbf{M}} d_m(a_{ik})] * [n_p(a_i)] \right\} \quad (1)$$

$$; n_p(a_i) = 1 \leftrightarrow a_i \neq a_{Jump}$$

Where h_{dif} is the overall difficulty score of the challenge.

a_i is a player's action.

d_p is the difficulty score based on the player's skill for the given action.

d_e is the difficulty score of the enemy involved in the player's action.

d_m is the difficulty score of the map object used during the player's action.

n_p is the number of times the player's action is repeated continuously. Currently, only jumping generates the score. Otherwise, it is 1.

a_{ij} is the property of the enemy involved with the player's action.

a_{ik} is the property of the map object used during the player's action.

A is the set of basic actions that can be carried out by the player.

E is the set of enemies' properties.

M is the set of map objects' property.

A chromosome represents one challenge. A level can have any number of challenges from a challenge group. In our prototype, a level consisted of two challenges from each challenge group. At the first level, chromosomes were created so that each of its actions did not exceed their default difficulties values. For each challenge group, its Challenge Rank score (CR) for a player could be calculated from the player's performances during play. The value of CR for a group of challenges was used to calculate the change in difficulty score for the challenges of that group in the next level. Equation (2) - (4) show how CR was calculated.

$$RL = (PT + 2) - (ST + 1) \tag{2}$$

$$PR = (2 * RL) - 1 \tag{3}$$

$$CR = PT - PR \tag{4}$$

Where PT is the number of times the challenges in the group were attempted.

ST is the number of times the challenges in the group were overcome.

RL represents Rank Level. It is the number of character used for the challenges in the group over several plays. Its smallest value is 1, which happens when the player character did not die at all.

PR is Play Rank. It is actually RL rescaled in order to calculate CR.

CR is Challenge Rank. It is transformed from RL so that scores are given more to the number of successful plays.

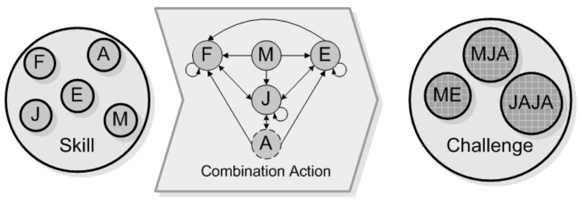


Figure 1. Challenge Generation

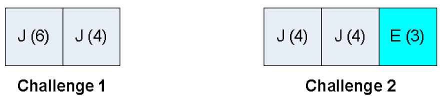


Figure 2. Chromosomes contain actions and their difficulty scores

In our prototype, after a 5th level was played, PT and ST only counted the last three levels. A chromosome consisted of a string of action types and their corresponding difficulty scores. Figure 2 shows two chromosomes. Both were from the same challenge group, JJ (this group contained challenges that started with two consecutive jumps).

3.1 Chromosome Preparation

We produced extra chromosomes to be used for crossover with original chromosomes. Each challenge group was used to generate a number of extra chromosomes. In our prototype, the number of generated chromosomes was the same as the number of original chromosomes. The CR value from a challenge group was used to construct extra chromosomes for that group. If CR >0, a newly produced chromosome should have a higher difficulty score than its source. However, the difference in scores should not exceed a factor of CR (such factor could be adjusted). If CR < 0, the new chromosome should have a lower difficulty score than its source, but the difference should not exceed the value of CR. For each challenge group, extra chromosomes were produced according to the following steps.

1. The first few elements in the chromosome that identified the group were generated according to that group's identity.
2. Then
    a. If CR >0, a legal action was randomly added to the chromosome.
    b. Else If CR <0, the new chromosome built so far was compared with its source. If the difficulty score of the new chromosome was less than its source, an action would be added to the chromosome.
3. The chromosome built so far was then checked to see if its difficulty score matched the value that was needed. Equation 6 and 7 were utilized.

$$h_{dif}(cnd) = h_{dif}(n) - h_{dif}(p) \tag{6}$$

$$LF = CL * CR(n) \tag{7}$$

Where $h_{dif}(n)$ is the difficulty score of the chromosome built so far.

$h_{dif}(p)$ is the difficulty score of the chromosome used as source.

LF is Learning Factor.

CL is Coefficient of Learning. It is a value used to scale the CR value.

$CR(n)$ is the difficulty level the current player could overcome.

If the challenge group CR value of the current level is greater than 0, the value of $CR(n)$ will be between the CR value of the previous level and the CR value of the current level. If the challenge group CR value of the current level is less than 0, the value of $CR(n)$ will be equal to the CR value of the current level.

Then

a. If $h_{dif}(cnd)$ < LF , the new difficulty score had not reached the value suitable for the player. Step 2 was then revisited.
b. If $h_{dif}(cnd)$ > LF , the new difficulty score was too high. The difficulty score of every action in the chromosome was checked. If every action had its least possible score, nothing would be done and the algorithm proceeded to the crossover. Otherwise, an action that had a lower difficulty score and was situated nearest to the end of the chromosome was chosen. Its difficulty score was then reduced. Then step 3 was repeated again.
c. If $h_{dif}(cnd)$ = LF , the player should be able to play the new challenge and challenges generated from it. The crossover was performed next.

3.2 Crossover

Our crossover differed slightly from standard Uniform crossover. Parts of the chromosomes which identified their challenge groups were not modified. Furthermore, each resulting chromosome from our approach needed to be checked for correct continuous actions (see figure 1). After the crossover, chromosomes were selected from the results. The chosen chromosomes would become the challenges of the next level. In our prototype, we selected chromosomes with the value $|h_{dif}(cnd) - LF|$ between 0 and 1. The reason we had to allow other values apart from 0 was because there might not be any chromosome with $|h_{dif}(cnd) - LF| = 0$ at all after the crossover. We selected the ones with $|h_{dif}(cnd) - LF|$ nearest to 0 first. If more than one chromosome had equal marks, their order was randomly chosen.

4. Testing and Results

Twelve testers were asked to play our game twice. In the first play, the game utilized Kamnerdnond's level generation methodology. In the second play, our level generation technique was applied. A tester cleared 20 levels for each play. During each player's session, the difficulty score and the number of lives the player spent for each challenge were recorded. The memory usage data for each level was also collected. After finishing both games, each player was asked about how he felt when playing each game and how the game difficulty changed during play. We need the following model behavior. First, when a player spent many lives overcoming a challenge, challenges of the same group in the next level must become easier (but not too easy). Second, when a player spent no life or very few lives overcoming a challenge, challenges of the same group in the next level must become harder (but not too hard).

Due to limited space, we cannot show results obtained from every player. However, all the players' results were very similar. Figure 3 shows the average difficulty score of each challenge group for one of the players during his 20-level-play of our model. Lives spent by the player in figure 3 are displayed in figure 4. Table 1 shows each challenge and lives spent to overcome it by the same player.

From the figure 3, 4 and table 1, it can be seen that when a player spent many lives for a group of challenges in a single level or spent some lives for the same challenge group over consecutive levels, the challenge difficulty score for that group tended to go down in the next level. On the other hand, if the player rarely died, the challenge difficulty score for that group tended to go up quickly. Only very few challenges did not follow this behavior. The players' opinions support this conclusion. Ten out of twelve players (83.33%) felt that after they encountered very difficult challenges in a level, the next level became easier. Eight out of twelve players (66.67%) felt that after they played a very easy level, the next level became more difficult. All the results indicate that our proposed model can effectively adjust the game difficulty level according to players' performances.

Figure 5 displays the average memory usage of each level from Kamnerdnond's model and our model, collected from 12 testers, each playing 20 levels for each model. Using the paired t-test, it was found that the two-tailed P value was less than 0.0001. By conventional criteria, this difference was considered to be extremely statistically significant. The difference between the mean of Kamnerdnond's model and our model equaled 160972.7995. The 95% confidence interval of this difference was from 152089.6254 to 169855.9736. The intermediate values used in calculations included t = 37.9279, df = 19 and standard error of difference = 4244.180. Therefore, it can be statistically shown that our new model has better memory utilization.

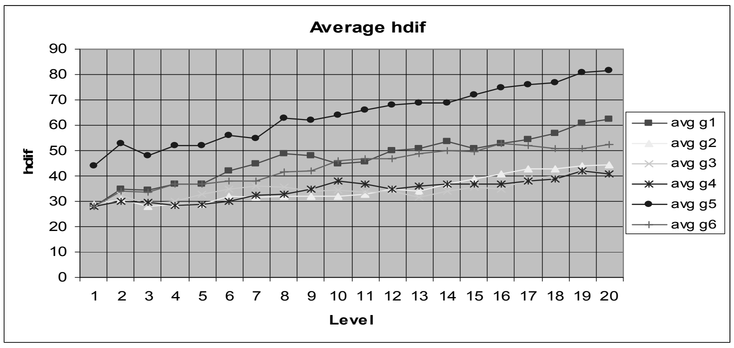


Figure 3. Average level difficulty score of player A for each challenge group

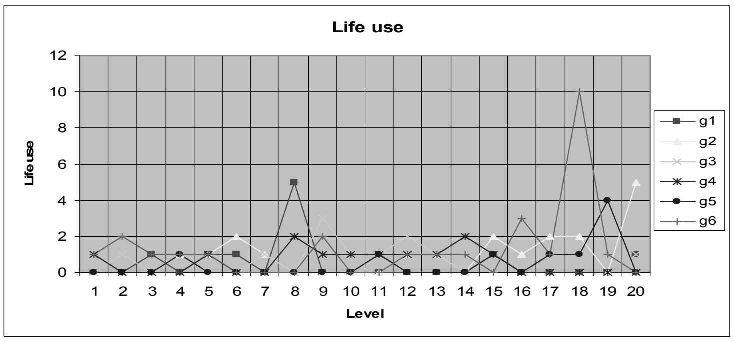
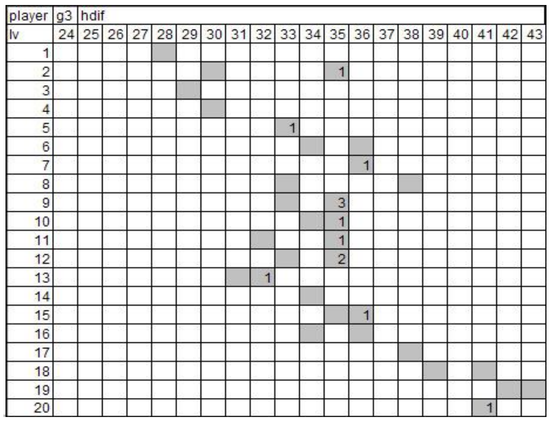


Figure 4. Lives spent by player A for each challenge group

Table 1. Lives spent by player A for each challenge in group 3



player	g3	hdif																		
lv	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
1					▨															
2							▨					▨ 1								
3						▨														
4							▨													
5										▨ 1										
6											▨		▨							
7													▨ 1							
8										▨					▨					
9										▨		▨ 3								
10											▨	▨ 1								
11									▨			▨ 1								
12										▨		▨ 2								
13								▨	▨ 1											
14											▨									
15												▨	▨ 1							
16											▨		▨							
17															▨					
18																▨		▨		
19																			▨	▨
20																		▨ 1		

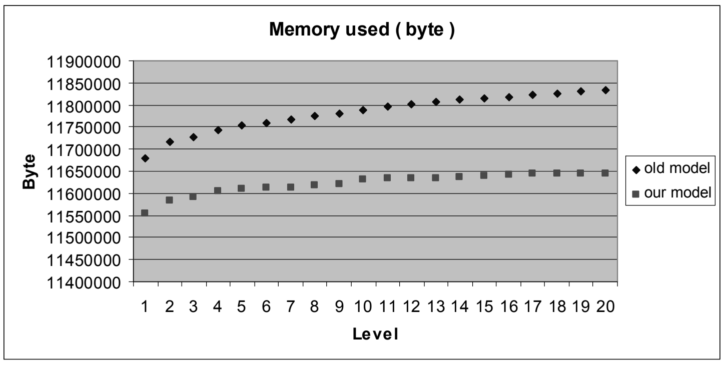


Figure 5. Average memory usage

Conclusion

Our main contribution was the new level generation methodology for platform games inspired by genetic algorithm. Levels generated with our methodology had their difficulty that suited each player's skill. The model also had lower memory usage compared to the reinforcement learning approach. There was some problem with the random nature of crossover. Sometimes a crossover did not produce any good results. However, this was very rare.

References

[1] Chariya Kamnerdnond and Vishnu Kotrajaras. Automatic Level Difficulty Adjustment in Platform Games Based on Player's Performance: Super Mario Case Study, *In Proceedings of the 11th National Computer Science and Engineering Conference*, Bangkok, Thailand. : 223-229. 2007.
[2] Tom M. Mitchell. *Machine Learning*, McGraw – Hill, International Edition 1997, ISBN: 0-07-042807-7.
[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning (An Introduction)*, The MIT Press, Cambridge, Massachusetts, London, England, 1998, ISBN: 0-262-19398-1.
[4] Sander Bakkes, Pieter Spronck, and Eric Postma. TEAM: The Team-oriented Evolutionary Adaptability Mechanism, In Matthias Rauterberg, editor, *Entertainment Computing - ICEC 2004*, Lecture Notes in Computer Science, Springer-Verlag, 3166: 273–282. 2004.
[5] Nicholas Cole, Sushi J. Louis, and Chris Miles. Using a Genetic Algorithm to Tune First-Person Shooter Bots, *Congress on Evolutionary Computation 2004*. 1:139-145. 2004.
[6] Thore Graepel, Ralf Herbrich, and Julian Gold. Learning to Fight. *In Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*. :193-200. 2004.
[7] Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen-Kuyper, and Eric Postma. Adaptive Game AI with Dynamic Scripting. *Machine Learning*. 63(3): 217-248. 2006.
[8] Gustavo Andrade, Geber Ramalho, Hugo Santana, and Vincent Corruble. Challenge-Sensitive Action Selection: an Application to Game Balancing, *In Proceedings of the 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'05)*. : 194-200. 2005.
[9] Randy J. Pagulayan, Kevin Keeker, Dennis Wixon, Ramon L. Romero, and Thomas Fuller. User-centered Design in Games, *Handbook for Human-Computer Interaction in Interactive Systems*, Microsoft Corporation, Mahwah, NJ: Lawrence Erlbaum Associates, Inc. 2003.
[10] Staffan Björk and Jussi Holopainen. *Patterns in Game Design*, Charles River Media, Inc., 2005, ISBN: 1-58450-354-8.