EVOLVING FUZZY LOGIC RULE-BASED GAME PLAYER MODEL FOR GAME DEVELOPMENT

Vorachart, Varunyu
Graduate School of Design, Kyushu University

Takagi, Hideyuki
Faculty of Design, Kyushu University : Professor

出版情報：International Journal of Innovative Computing, Information and Control. 13 (6), pp.1941-1951, 2017-12
バージョン：
権利関係：

EVOLVING FUZZY LOGIC RULE-BASED GAME PLAYER MODEL FOR GAME DEVELOPMENT

Varunyu Vorachart[1] and Hideyuki Takagi[2]

[1]Graduate School of Design
Kyushu University
Fukuoka, Japan 815-8540
varunyu@kyudai.jp

[2]Faculty of Design
Kyushu University
Fukuoka, Japan 815-8540
http://www.design.kyushu-u.ac.jp/~takagi/

Abstract. *We propose a framework for automatic game parameter tuning using a game player model. Two kinds of computational intelligence techniques are used to create the framework: a fuzzy logic system (FS) as the decision maker and evolutionary computation as the model parameter optimizer. Insights from a game developer are integrated into the player model consisting of FS rules. FS membership function parameters are optimized by a differential evolution (DE) algorithm to find optimal model parameters. We conducted experiments in which our player model plays a turn-based strategy video game. DE optimisation was able to evolve our player model such that it could compete well at various levels of game difficulty.*
Keywords: player model, fuzzy logic system, differential evolution, video game

1. **Introduction.** Making games fun is one of the most important objectives in video game development. A game should not be so difficult that it discourages a beginner, nor so easy that it bores a skilled player [1]. There are many factors involved in creating game engagement. One way to engage the player is to properly adjust the game parameters. Game parameters are variable settings in a video game that control the game difficulty. However, it is not easy to manually tune game parameters to make an enjoyable game. Game designers must iteratively tweak the game parameters to achieve a design plan [2]. This is a laborious and time-consuming task. With a limited production schedule, game designers need new tools to speed up the game parameter tuning process.

Our goal is to design a methodology for helping game developers by fine-tuning game parameters automatically. At the same time, the system should allow game developers to easily use their game expertise to assist in the tuning process.

As the first step toward the automatic tuning of game parameters, this paper proposes a game player model to be used for game development. A game player model is a computational model which represents the interactions between a human game player and the video game environment from a specific point of view. In a broader sense, the form of the representation can be cognitive, affective, or behavioral [3]. We focus on the player's decisions for the game's input. Hence, a human game player is replaced with our player model as shown in Fig. 1. Subsequently, we let the model play the game repetitively with different settings and variations to automatically search for desirable game parameters.

Our game player model uses computational intelligence approaches for knowledge-based modeling and optimization. Similar to pioneering work on improving fuzzy logic systems (FS) for process control with genetic algorithms [4], our system consists of a FS to express the insights from a skilled game player, and evolutionary computation (EC) to search for the optimal FS parameters. We expect that our model should evolve and produce better

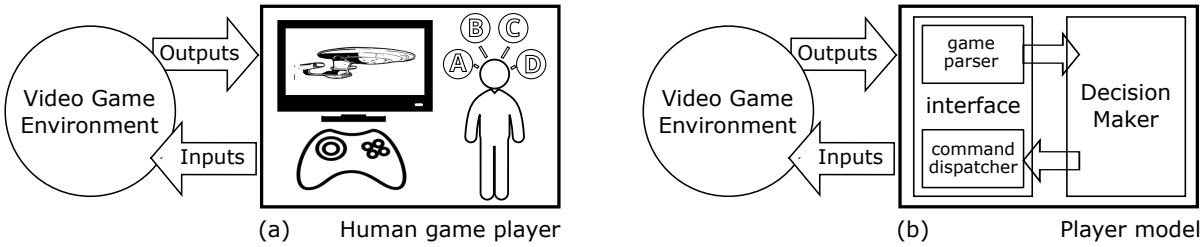


FIGURE 1. (a) A human game player is replaced by (b) a player model.

performance than if modified manually by an expert game developer, thanks to EC, and confirm this result in this paper.

Following this section, we provide a background on some game player models and related topics in section 2. Our proposed player model is explained in section 3. Section 4 shows our experiment setup, results and discussions thereof. Finally, our conclusions and plans for future works are presented in section 5.

2. Background & Related Topics.

The role of player models is to either manipulate a game itself or to inform game designers about a game player's experiences or behaviors [5]. However, modeling a game player to assist video game designers, which is also known as designer modeling, is still in its early stages [3]. This type of player model can be viewed as a second-order player model [5]. That is, a player's behaviors simulated by the player model are evaluated with the intention of aiding the game designer.

'Game player model' is an umbrella term for how to represent specific information when a game player interacts with a video game. There are many kinds of player models with different intended purposes, scopes of application, sources of derivation, and domains of finding [6]. Our model can be described as an *individual analytic action generator* following the terminology of Smith et al. [6]. This is a model that generates game inputs for an individual player from an automated method that examines the mechanisms of a game play. In our case, a FS is used as the analytical tool to generate the player's actions.

Due to its simple, yet expressive and powerful algorithm, FS is a very popular artificial intelligence approach in video games (game AI), both commercially and academically [7]. With relatively less effort to incorporate into an existing game design, FS has been used in commercial video games, typically in the form of a fuzzy state machine. Many successful games in various genres — for example, a first-person shooter like Unreal, a turn-based strategy game like Civilization: Call to Power, and a simulation game like The Sims — have applied fuzzy logic to their AI mechanisms [7]. For academic game AI, FS is often implemented as an extension to other computational intelligence techniques forming more-advanced game AI methods. Examples are fuzzy neural networks, fuzzy Q-learning, and the evolutionary fuzzy systems that our player model is based on.

An evolutionary fuzzy system is a hybrid system between EC and FS. It incorporates the linguistic expressivity of FS and the adaptive learning capabilities of EC. Many game AI researchers have applied EC techniques with FS to control game characters. For instance, an evolutionary strategy for a Pac-Man game [8], coevolution [9] and a genetic algorithm [10] for a racing game. Previously, evolutionary fuzzy systems have been used generally to manipulate game characters. However, in our automatic game parameter tuning, it is developed as a tool for the game designer instead.

3. Framework for Game Player Model.

3.1. Interaction with Video Game Environment.

We design our player model based on the interaction between a video game player and a video game environment. We consider the video game environment as a set of game states with three-actions loop: input,

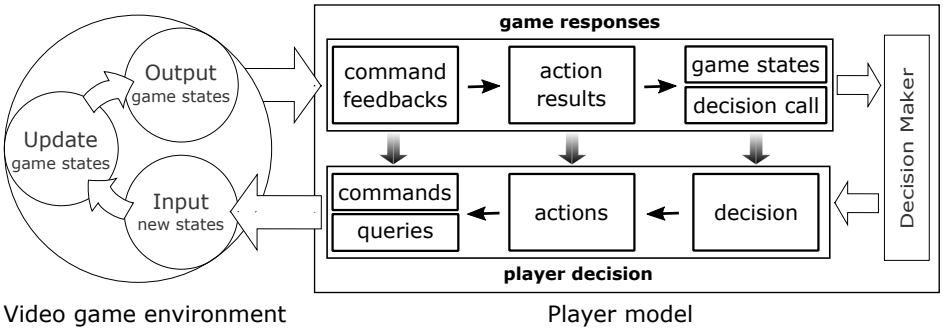


FIGURE 2. Data and information flow between a video game environment and our player model.

update, and output. A game receives commands from a game player to change its states via game inputs. The game then updates its states and responds back to a game player via game outputs. Therefore, a player receives game responses and makes a decision accordingly. Later on, the player transforms the decision into game commands and responds back as game inputs. The game loop continues iteratively until the game ends. The relationship between player decisions and game responses establish the foundation of our player model's framework.

3.1.1. *Hierarchy of Player Decision.* We model a player decision while playing a video game in a top-down approach, from an abstract idea down to concrete game commands. This hierarchy of the player's decisions consists of a decision, an action, and a game command. First of all, a strategic plan to achieve the game's objective yields a target decision according to the current game states. Next, the player constructs a series of actions to accomplish the target decision. Then, the player inputs game commands along with their corresponding parameters to control the game according to the decided actions.

3.1.2. *Levels of Game Responses.* In response to player inputs, a video game sends out data that can be classified as game states and game information. Game states represent the current value of variables in the game environment, and game information is the reaction from the game to the corresponding decision hierarchy. Depending on this information, the subsequent game command is either a follow-up to the preceding ones or a call for a new decision.

We classified game information with a bottom-up approach, from command feedback up to decision calls. The levels of game information therefore consist of command feedback, action results, and decision calls. First, a command feedback is a quick response made to game command inputs or their parameters. It would include e.g. a notice that a command was invalid or give the expected range of a command parameter. Second, an action result is when the game state is updated in response to an issued action command. This indicates progress towards or recess away from a player's target decision. Finally, a decision call is a request for a new decision which will be consumed as game input. A special kind of decision call, namely a critical call, must be noted [11]. This call requires an immediate resolution from the player. Without a suitable response, the player's game states may become worse and the player may even lose the game.

3.2. **Game Player Model.** The association between the hierarchical levels of player decisions and the matching game information is used as the underlying mechanism for our game player model, as presented in Fig. 2. Output from a video game environment is parsed by a game interface and supplied to a decision maker to generate a player decision. Later, a game command generated from this decision is dispatched back as an input to the game. Each module is discussed in more detail in the following subsections.

3.2.1. *Game Interfaces.* Our player model communicates with a game environment through a game interface module, as shown in Fig. 1 (b). The interface mainly consists of a game parser and a command dispatcher. A game parser extracts necessary game information and game states from game outputs. Depending on the video game, the output format may be textual, visual, auditory, haptic, etc. Game information is used to evaluate issued actions or commands while game states are supplied to a decision maker. Essential game states are stored internally to keep track of past game states. The ability of a player model to store a large number of past game states is an advantage over a human player's restricted memory.

Afterward, when the game requests an input, a command dispatcher delivers inputs to the game environment in a suitable format. In general, there are two types of game inputs: an action command that modifies a game state and a query command that retrieves the current game state. For a player to make a correct decision in a game, a necessary amount of game states must be gathered carefully with a certain degree of accuracy either by querying the game directly or deducing them from other game states implicitly,

3.2.2. *Decision Maker.* The brain of our player model is the decision maker module. Game player experts express their decision-making knowledge linguistically with fuzzy logic rules (FS rules). This requires less effort for non-technical professionals to present their expertise in the domain. The main advantage of FS is that it is easy to create, understand, and maintain. In addition, new findings may be discovered from the optimized FS parameters due to their high level of interpretability.

The FS consists of three major components as illustrated in Fig. 3: FS rules, membership functions, and the FS reasoning engine. First, the FS rules specify the relationship between game states, in the form of FS inputs and Boolean inputs, and their consequent game decisions. The game developer provides knowledge of game decisions with FS rules. Fig. 4 shows an example of FS rules implemented in our experiments. Second, the membership function is an interpretation of the FS variable used in the rules. The function is governed by function parameters that define the shape of the function and lead to a certain resulting decision. Membership function parameters can be assigned manually by a skilled game player or optimized automatically by an optimization algorithm. Lastly, the FS reasoning engine examines current values of game states according to the supplied FS rules and membership functions and calculates player decisions accordingly.

Designing FS rules is a demanding task for a novice. Three features have been developed to aid in the creation of FS rules: a modular FS table, multiple-output decisions, and FS table re-evaluation. The modular FS table reduces table size drastically by dividing FS rules into smaller tasks. Smaller tables are easier to create and maintain than larger and complicated ones. Multiple-output decisions allows many decisions to be expressed via a single rule. Depending on the design of player's behaviors, a game developer is able

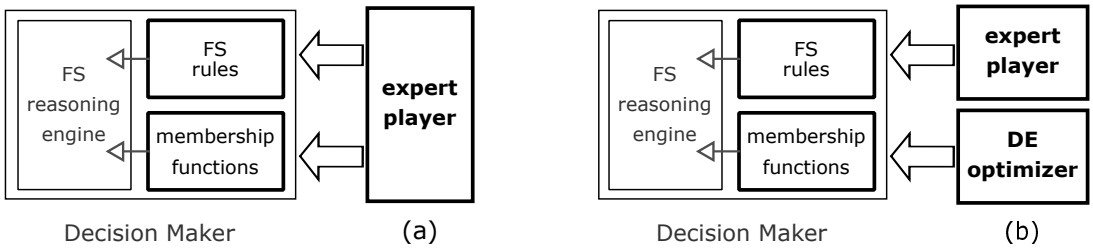


FIGURE 3. The decision maker in our player model. While an expert player uses his or her skills to create FS rules, the FS membership function parameters are tuned by either (a) an expert player or (b) a DE optimization algorithm.

to provide optional decisions and let differential evolution (DE) optimization find the right decision instead. FS table re-evaluation activates automatically when a FS decision generates an invalid command feedback due to unexpected game events or mistaken game information retrieval. In this case, the FS will be re-evaluated with the false decision disabled. This ensures a new decision will be made by the FS reasoning engine.

In our implementation, the output decision from the FS reasoning engine is specified in the form of a decision function. The decision function is called to generate a series of actions to achieve the target decision. Each action is then mapped to one or more game commands with their corresponding parameter(s). Command parameters are computed by a certain formula or optimized as preset values by DE optimization.

3.2.3. *Player Model Parameter Optimization.* We select key parameters from the decision maker discussed in the previous subsection, namely the FS membership parameters, multiple-output decision parameters, and the preset command parameters, as player model parameters. In our experiments, these parameters are adjusted manually by a skilled game player or optimized automatically by a parameter optimization algorithm.

Although, in general, any population-based optimization method could be used, we select a DE algorithm to optimize our player model's parameters. The main advantages of DE are its simplicity and ease of use. Moreover, DE is efficient for optimizing real-valued, multi-modal problems. It yields good convergence properties and performs well at avoiding local optima.

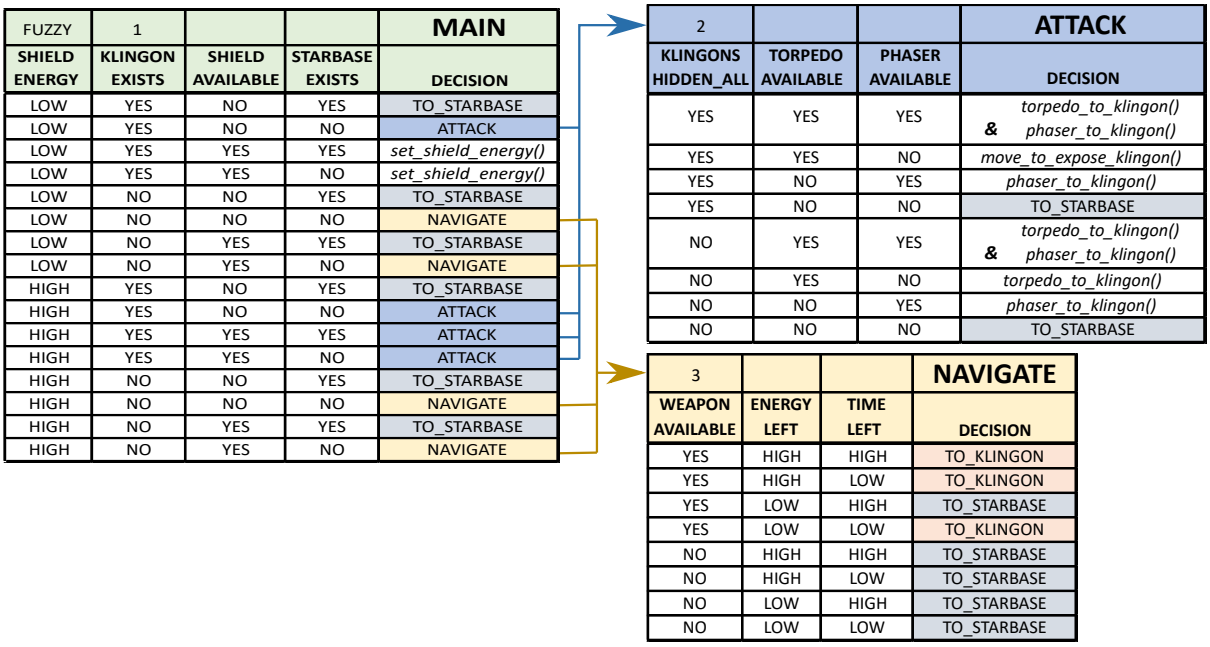
We use DE optimization in several parts of our player model. The primary use is to optimize FS membership parameters for decision making. During the FS inference process, a FS membership function maps a FS input variable and its degree of membership to a numerical value. With different sets of FS membership parameters, various numerical interpretations of the fuzzy terms are achieved and result in different game decisions. Additionally, we use DE optimization to search for the best multiple-output decision in a FS rule. For this, the rule weight is divided proportionally according to the weight given for each distinct output by the DE optimizer. The divided rule weights are added to the output weight of the corresponding output decisions. The output with the maximum weight is selected as the FS decision. This decision may point to another FS table for further reasoning computation or call a decision function for action planning and game command generation. Furthermore, we assign the value of a gene element to the preset value of command parameter. The preset values are used directly as command parameters or indirectly as numerical constants for some calculations.

While playing a game, the player model parameters determine decisions which generate all game actions. These generated actions interact with the video game environment and a variety of game states emerge. At the end of the game, selected game states are evaluated as a fitness score in the search for the population's best DE individual, representing the best player of the game.

4. **Experiments of Evolving Game Player Model.**

4.1. **Star Trek Game.** We use Star Trek, a turn-based text strategy game [12], as a test bed for our player model. A player controls the Starship Enterprise to survey the 8×8 quadrants of an unexplored galaxy. The objective is to destroy all Klingon spaceships within a given game time. Two kinds of weapon can be used for an attack: photon torpedo or phaser. A player has to manage the energy for navigation, attack, and shield. The energy can be recharged at one of the Starbases located throughout the galaxy.

Following from the game play outlined above, there are three game parameters for controlling game difficulty: (1) the game time limit for destroying all Klingons, (2) the number of Klingons, and (3) the number of Starbases. Among these parameters, the game time has a direct effect on the game difficulty. The less game time, the greater the game



FUZZY	1			MAIN
SHIELD ENERGY	KLINGON EXISTS	SHIELD AVAILABLE	STARBASE EXISTS	DECISION
LOW	YES	NO	YES	TO_STARBASE
LOW	YES	NO	NO	ATTACK
LOW	YES	YES	YES	*set_shield_energy()*
LOW	YES	YES	NO	*set_shield_energy()*
LOW	NO	NO	YES	TO_STARBASE
LOW	NO	NO	NO	NAVIGATE
LOW	NO	YES	YES	TO_STARBASE
LOW	NO	YES	NO	NAVIGATE
HIGH	YES	NO	YES	TO_STARBASE
HIGH	YES	NO	NO	ATTACK
HIGH	YES	YES	YES	ATTACK
HIGH	YES	YES	NO	ATTACK
HIGH	NO	NO	YES	TO_STARBASE
HIGH	NO	NO	NO	NAVIGATE
HIGH	NO	YES	YES	TO_STARBASE
HIGH	NO	YES	NO	NAVIGATE

2			ATTACK
KLINGONS HIDDEN_ALL	TORPEDO AVAILABLE	PHASER AVAILABLE	DECISION
YES	YES	YES	*torpedo_to_klingon()* & *phaser_to_klingon()*
YES	YES	NO	*move_to_expose_klingon()*
YES	NO	YES	*phaser_to_klingon()*
YES	NO	NO	TO_STARBASE
NO	YES	YES	*torpedo_to_klingon()* & *phaser_to_klingon()*
NO	YES	NO	*torpedo_to_klingon()*
NO	NO	YES	*phaser_to_klingon()*
NO	NO	NO	TO_STARBASE

3			NAVIGATE
WEAPON AVAILABLE	ENERGY LEFT	TIME LEFT	DECISION
YES	HIGH	HIGH	TO_KLINGON
YES	HIGH	LOW	TO_KLINGON
YES	LOW	HIGH	TO_STARBASE
YES	LOW	LOW	TO_KLINGON
NO	HIGH	HIGH	TO_STARBASE
NO	HIGH	LOW	TO_STARBASE
NO	LOW	HIGH	TO_STARBASE
NO	LOW	LOW	TO_STARBASE

FIGURE 4. Three out of five modular FS tables used in Star Trek game. Two FS rules in the ATTACK table (top right) use the multiple-output decision feature. Their are indicated by the "&" symbol in the decision column.

difficulty regardless of how the galaxy map is initialized. The difficulty also depends on the number of Klingons and their distribution throughout the galaxy. In general, the more Klingon spaceships, the greater the game difficulty. Finally, the number of Starbases has less effect on the game difficulty because it is not directly related to the game's objective: destroy all enemies within a limited time. Hence, we adjust only game time and the number of Klingons to control the game difficulty in our experiments.

4.2. **FS Engine & DE Optimization.** We obtain input data to the FS decision maker by parsing screen text from the game output. Thus, the player model perceives the same information as a human player. This ensures that all decisions are made according to the game outputs, not data which is internal to the game environment.

For the FS decision maker, we create five modular tables (i.e., MAIN, ATTACK, NAVIGATE, TO_STARBASE, and TO_KLINGON) consisting of five binary FS inputs (LOW or HIGH) and nine Boolean inputs. Selected FS tables and their modular relationship are shown in Fig. 4.

For FS inference by membership function, we implement a triangular-shaped membership function for its quick computation and minimum number of function parameters. The shape of each membership function is determined by the gene elements in DE population individuals, as presented in Fig. 5 (a). The height of each triangle membership function is exactly one. At any position along the x-axis, the summation of overlapped membership functions always equals to one. To represent a series of triangular functions, one gene element serves as the position along the x-axis where the function shape reaches its peak. This same position is also a base, at zero height, of the preceding and following triangle. Hence, two gene elements represent a membership function with two degrees of membership: LOW and HIGH and three gene elements represent a membership function with three degrees of membership: LOW, MEDIUM, and HIGH, and so on. A minimum and maximum value for each FS variable are specified by an expert player and the DE algorithm fits the optimized parameters within this range.

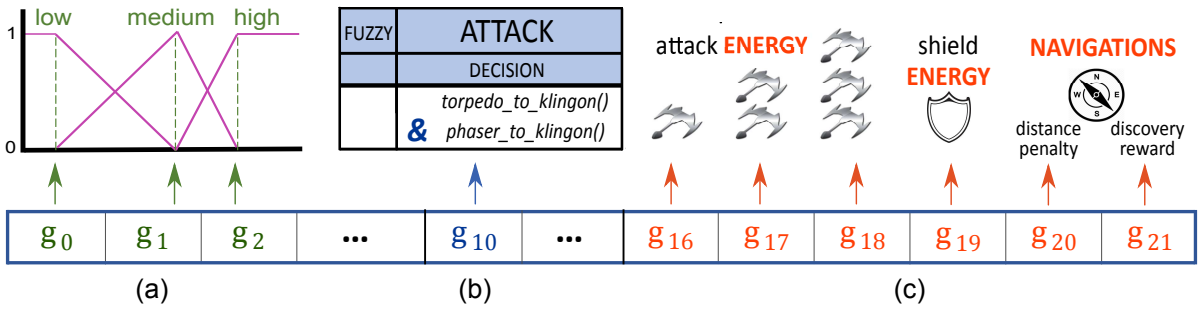


FIGURE 5. Parameter coding in DE population individuals. We implemented our DE optimizer to search for (a) optimal triangular shapes for FS membership functions, (b) optimal decisions for the multiple-output FS rules, and (c) optimal preset values for command parameters.

In addition to membership parameter optimization, we use DE to optimize the multiple-output decisions in the FS rules and the preset values of some command parameters. Both features are illustrated in Fig. 5 (b) and (c) respectively. In multiple-output optimization, each gene element represents the weight of an output decision. When calculating the rule weight, the optimized output weights split the specific rule weight for each output proportionally. In preset command parameter optimization, the gene element is directly used as a constant for the specific command in the whole game. For example, when the FS decision is phaser_to_klingon() and there is only one Klingon in a quadrant, the player model will fire phasers with an energy of g_{16}.

For the DE algorithm environment, we set the scale factor and crossover rate to 0.9 and 0.8, respectively. Our DE employs the DE/best/1 mutation scheme and uses a binomial crossover operator. We provide 40 DE populations to optimize 22 model parameters: 10 FS membership function parameters, 2 multiple-output decision parameters, and 10 preset command parameters. Each individual in the population is represented by an array of 22 real numbers.

For a specified number of Klingons, we initialized ten galaxy maps with different Klingon distributions. The map distribution is incremental: a new Klingon is placed in a random location while the existing Klingons' locations are unchanged. With each setting of the game parameters (i.e., game time and the number of Klingons), each individual of the population plays the same set of ten Star Trek games with different initial galaxy maps.

$$fitness\ score = average\ game\ scores + win\ ratio \tag{1}$$

$$\begin{aligned} game\ score = {} & 0.60 * \#destroyed\_enemies \\ & +0.30 * \#found\_enemies \\ & +0.10 * (win\_game\ ?\ \#remaining\_time : 0.0) \end{aligned} \tag{2}$$

$$win\ ratio = \frac{\#win\_games}{\#total\_games} \tag{3}$$

The fitness function, shown in Eq.(1), is an average of the scores of ten games plus a win ratio incentive. The objective of the fitness function is to win more games or, if unavoidable, lose by as little as possible. The game score in Eq.(2) is calculated by the weighted sum of three normalized final game states. First, the number of destroyed Klingons is selected to motivate destroying more Klingons, which eventually leads to winning a game. Hence, the largest weight is assigned to this number. Second, the number of Klingons found in the galaxy is chosen to encourage better exploration; we cannot win a game unless all Klingons are discovered. Finally, the remaining time is used to distinguish the best winner from average winners. When winning the same game, the faster win is preferable. A win ratio, shown in Eq.(3), is a motivation to win more games.

4.3. **Simulation Setups.** We simulated our player model at various levels of game difficulty. Each game difficulty is obtained by altering two game parameters: the number of Klingons and the game time. Increasing the number of Klingons or decreasing the game time result in more difficult games. The number of Starbases was fixed at three due to its diminished impact on game difficulty. We characterize game difficulty into the three levels of easy, medium and hard. For each game parameter, we choose three values at an equal interval that is large enough to distinguish between the three levels of game difficulty.

Klingon spaceships are randomly distributed throughout 64 quadrants. To give a suitable occupation rate, we use settings of 10, 15, and 20 Klingon spaceships for easy, medium and hard. If the Enterprise is in perfect condition, it takes 20 Stardates to survey the entire galaxy. Thus, we determine that 40 Stardates is a moderate time to complete the games mission, so settings of 30, 40 and 50 are used for hard, medium and easy. The combination of the two game parameters settings establishes the total number of game difficulty parameters sets as nine.

We conducted 18 sets of simulations to evaluate the tuning capabilities of our player model: nine for DE-optimized tunings and another nine of the same game parameter setups for manual tunings. For DE-optimized tunings, we ran the DE simulation 50 times with different initial populations. Each individual population plays ten Star Trek games with unique galaxy maps. For manual tunings, we created three sets of player model parameters to play against 10, 15 and 20 Klingons at all three game times (30, 40, and 50 Stardates) for the same set of ten Star Trek galaxies once and then compared the results with the DE-optimized tunings.

4.4. **Results & Discussions.** For each set of simulations, we plotted the best fitness scores and the corresponding maximum numbers of games won in Fig. 6 (a) and (b), respectively. The left side of both graphs shows the nine fitness scores from the manual tunings, and the right shows the evolution of the nine averaged best fitness scores in 50 runs from DE optimizations. All DE-optimized tunings reached their stable states within 200 generations. All optimized player model parameters performed better than the manual tunings with the corresponding game parameters in all 50 runs. The only exception is with the most difficult game settings, with the maximum number of Klingons

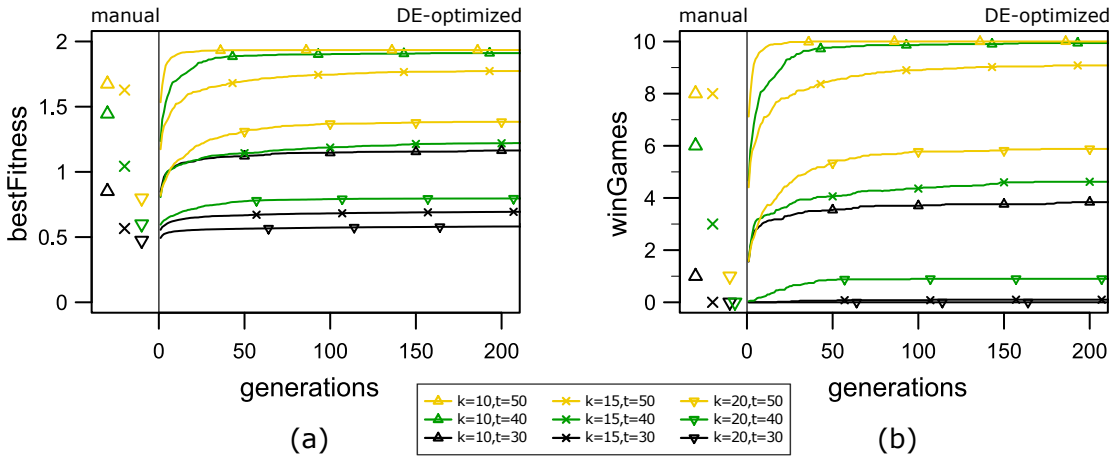


FIGURE 6. (a) The best fitness score and (b) the corresponding maximum numbers of games won from manual tunings (points graph) and DE-optimized tunings (line graph). We conducted experiments with nine game parameter setups, which were obtained by varying the numbers of Klingons (k) and game time (t).

TABLE 1. Results from the Friedman test and the Holm's multiple comparison test. The symbols of $\ll$, $<$ and $\approx$ mean that there is a significant difference with significant level of 1%, 5%, and no significance, respectively. The subscripts of $_{\text{man}}$, $_{\text{1st}}$, and $_{\text{200th}}$ refer to the fitness values obtained by manual tunings, DE-optimized tunings at the 1st generation and at the 200th generation, respectively. Fitness values of DE-optimized tunings are the average of the best fitness scores from 50 trial runs.

game parameters	**Best fitness score**					**Number of won games (out of 10)**				
k=10, t=50	1.5338_{1st}	$\ll$	1.6741_{man}	$\ll$	1.9343_{200th}	7.12_{1st}	$\ll$	8_{man}	$\ll$	10.00_{200th}
k=10, t=40	1.2356_{1st}	$\ll$	1.4446_{man}	$\ll$	1.9108_{200th}	4.66_{1st}	$\ll$	6_{man}	$\ll$	9.94_{200th}
k=15, t=50	1.1759_{1st}	$\ll$	1.6276_{man}	$\ll$	1.7724_{200th}	4.42_{1st}	$\ll$	8_{man}	$\ll$	9.08_{200th}
k=20, t=50	0.7987_{man}	$\approx$	0.8173_{1st}	$\ll$	1.3838_{200th}	1_{man}	$\ll$	1.58_{1st}	$\ll$	5.88_{200th}
k=15, t=40	0.8070_{1st}	$\ll$	1.0430_{man}	$\ll$	1.2195_{200th}	1.62_{1st}	$\ll$	3_{man}	$\ll$	4.62_{200th}
k=10, t=30	0.8473_{1st}	$\approx$	0.8511_{man}	$\ll$	1.1640_{200th}	1_{man}	$\ll$	1.56_{1st}	$\ll$	3.84_{200th}
k=20, t=40	0.5950_{1st}	$<$	0.6000_{man}	$\ll$	0.7960_{200th}	0_{man}	$\approx$	0.04_{1st}	$\ll$	0.9_{200th}
k=15, t=30	0.5560_{1st}	$<$	0.5660_{man}	$\ll$	0.6931_{200th}	0_{man}	$\approx$	0_{1st}	$\ll$	0.1_{200th}
k=20, t=30	0.4740_{man}	$\ll$	0.4933_{1st}	$\ll$	0.5808_{200th}	0_{man}	N/A	0_{1st}	N/A	0_{200th}

(20) and the minimum game time (30 Stardates); here the best fitness improves a little and cannot win even a single game.

Table 1 compares the simulation results among three groups of parameter tunings: manual tunings, initial generation and stable generation of DE-optimized tunings. Initially, the manual tuning by an expert player performed significantly better than the 1st generation of DE optimization in six out of nine setups, especially in a group of easy games. However, within a few generations, the results from the DE optimization outperformed the manual tunings in all cases. This demonstrates that manual tunings by a skilled player played the game well to a certain degree. It is common for a population-based optimization to perform better than a human due to the size of its searching population. The optimization method also provides solution parameters with a higher degree of precision than can be achieved via human adjustment. Even though human experts can design the FS rules reasonably well, the interpretation of FS inputs with membership functions may not be an easy task.

From Table 1, we can clearly classify these nine game parameter setups into three difficulty levels. The first three rows are the easy-level games with an almost 100% winning rate and have the most optimization improvement. The last three rows are hard games with a winning rate less than 10% and hardly have any improvement at all. This is likely because the hard games have such a low winning rate at the start and therefore create a weak selection pressure to search for better solutions. On the contrary, the easy games have higher initial winning rates and generate a stronger selection pressure to search for optimal solutions.

5. **Conclusion.** The experiments demonstrated that our game player model was practical for competition in a turn-based strategy game with both manual and DE-optimized tunings. Thanks to DE optimization, our player model evolved reasonably well and the optimized model parameters generated better fitness scores than traditional expert tunings in all game-difficulty levels. The design of rule-based reasoning for decision making still requires the knowledge of a domain expert. The combination of a FS decision maker and an EC-based approach to parameter optimisation, which is not just limited to the DE technique, is a feasible framework for automatic game parameter tuning.

For future works, we need to improve the performance of our player model. Even though our optimized player model provides us with good results, its performance is still not comparable to expert players. When our player model is strong enough, we can

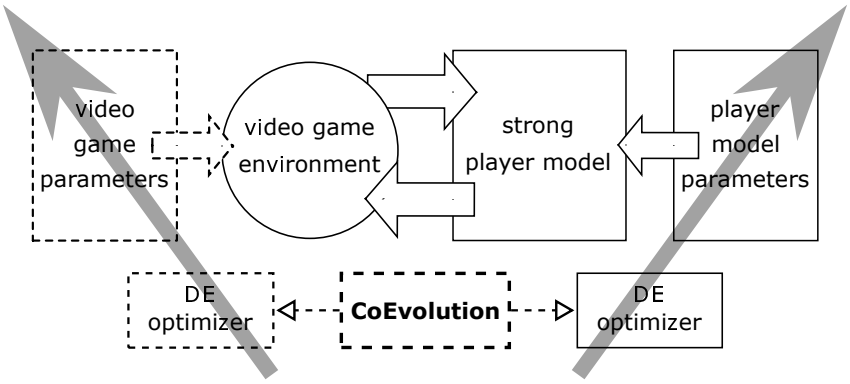


FIGURE 7. Future works are shown as dashed in the diagram. We will use a coevolutionary algorithm to evolve game parameters along with our proposed player model.

then use it to coevolve with the game parameters. As illustrated in Fig. 7, another EC optimization will evolve the game parameters and evaluate its population based on their counter interaction with the optimized game player model, and vice versa. Applying the competitive coevolution algorithm, our game player model will evolve to win as many games as possible for a given game difficulty while game parameters will evolve to search for more difficult game settings for the model. We hope that, with a robust player model, automatic game parameter tuning will become more practical for real-world applications.

REFERENCES

[1] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma, "Difficulty scaling of game AI," *Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAMEON'2004)*, Ghent, Belgium, pp. 33–37, 2004.

[2] M. O. Riedl and A. Zook, "AI for game production," in *IEEE Conference on Computatonal Intelligence and Games*, Niagara Falls, Canada, pp. 1–8, August 2013.

[3] G. N. Yannakakis and J. Togelius, "A Panorama of Artificial and Computational Intelligence in Games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 4, pp. 317–335, 2015.

[4] C. Karr, L. Freeman, and D. Meredith, "Improved fuzzy process control of spacecraft autonomous rendezvous using a genetic algorithm," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 1196, pp. 274–288, February 1990.

[5] A. Liapis, G. Yannakakis, and J. Togelius, "Designer Modeling for Personalized Game Content Creation Tools," in *9th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Boston, USA, pp. 11–16, 2013.

[6] A. M. Smith, C. Lewis, K. Hullett, G. Smith, and A. Sullivan, "An Inclusive View of Player Modeling," *Proceedings of the 6th International Conference on Foundations of Digital Games*, Bordeaux, France, pp. 301–303, June 2011.

[7] M. Pirovano, "The use of Fuzzy Logic for Artificial Intelligence in Games The current state of Game AI," *University of Milano, Milano, Italy*, 2012.

[8] H. Handa and M. Isozaki, "Evolutionary fuzzy systems for generating better Ms.PacMan players," in *IEEE International Conference on Fuzzy Systems*, Hong Kong, China, pp. 2182–2185, June 2008.

[9] S. Guadarrama and R. Vazquez, "Tuning a fuzzy racing car by coevolution," in *3rd International Workshop on Genetic and Evolving Fuzzy Systems, GEFS*, Witten-Bommerholz, Germany, pp. 59–64, Mar 2008.

[10] D. Perez, G. Recio, Y. Saez, and P. Isasi, "Evolving a fuzzy controller for a car racing competition," *IEEE Symposium on Computational Intelligence and Games*, Milano, Italy, pp. 263–270, Sep 2009.

[11] B. Vallade, A. David, and T. Nakashima, "Three layers framework concept for adjustable artificial intelligence," *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol. 19, no. 6, pp. 867–879, 2015.

[12] P. David, "Flights of Fancy with the Enterprise," *Byte Magazine*, vol. 2, pp. 106–113, Mar 1977.