Study on Computational Intelligence Approaches for Design of Game Strategies

ワランユー, ワラチャート

Doctoral Dissertation of Engineering

博士 (工学) 学位論文

Study on Computational Intelligence Approaches for Design of Game Strategies

ゲーム戦略設計のための計算知能に関する研究

2018 年 9 月

3DS14027R　　Varunyu Vorachart

Kyushu University

九州大学

Abstract

We propose a framework for automatic game parameter tuning using a fuzzy logic systems (FS) of a game player model and evolutionary computation (EC) that optimizes the FS and adjusts game parameters through simulations.

Controlling the difficulty levels of games is an important process in a game development. It is a crucial process to make the games attractive for a wider range of game players; too difficult games may discourage novice gamers, and too easy games may bore skilled gamers. Since the game difficulty is controlled by game parameters, game parameter tuning has to find the balance among game parameters that match a variety of gaming abilities.

Since a tuning process of game parameters is time-consuming and resource-intensive, especially on a tight schedule in video game production, it requires automated methodologies under the guiding directions of game developers.

Our proposed framework uses two computational intelligence techniques, FS and EC. A fuzzy rule-based game decision system, as a game player model, generates strategic decisions and game commands. Insights from a game developer can be integrated into the FS. We use an EC algorithm to optimize the parameters of the player model to make the model stronger. In our evaluation of the proposed framework, we focus on game parameter tuning in a turn-based strategy game and implement the framework into a turn-based strategy text game called *Star Trek*.

We use an FS-based player model as a simulated game player to playtest the game at various game parameter settings. To automate the playtesting, we suggest using a coevolutionary algorithm to evolve both player models and game parameters simultaneously. We expect that both the game and game player models mutually evolve each other until reaching the balance. In this environment, player models with ranges of gaming abilities may play a specific set of games equally well. Therefore, we believe that our framework can fine-tune game parameters computationally for a certain game difficulty that is suitable for game players with diverse skills.

概　要

ゲームプレーヤモデルのファジィシステム（FS）とそのFSを最適化する進化計算を用いてゲームパラメータの自動調整をする枠組みを提案し，シミュレーション実験で評価する．

難しすぎるゲームは初心者をめげさせてしまうし，易しすぎるゲームは熟練者を飽きさせてしまうため，ゲーム開発においてゲームの難易度を調整することは必須過程である．ゲームを幅広いプレーヤに対して魅力的にする必須の過程である．ゲームの難易度はゲームパラメータで制御されるため，ゲーム能力に合ったパラメータ間のバランスを取った調整が必要である．

ゲームパラメータの調整は時間もコストもかかり，特にビデオゲーム制作の厳しい開発スケジュールでは，開発者のガイドラインに沿った自動調整手法が求められている．

提案枠組みではFSとECの二つの計算知能技術を用いる．ゲームプレーヤモデルとしてのファジィルールベースのゲーム決定システムはゲーム戦略の決定とコマンドを生成する．ゲーム開発者の知見を組み込むことも可能である．ECはプレーヤモデルを強くするためのモデルパラメータ最適化に用いる．提案枠組みを評価するために，ターン制戦略ゲームのパラメータチューニングに焦点を当て，提案枠組みをターン制戦略テキストゲームであるStar Trekゲームに組み込む．

色々なゲームパラメータ値に設定したゲームをテストするため，疑似ゲームプレーヤとしてFSプレーヤモデルを用いる．ゲームパラメータ調整を自動化するために，共進化アルゴリズムで疑似プレーヤモデルとゲームパラメータを同時に調整することを提案する．共進化によって両者のバランスが取れるまでお互いが進化し合うことが期待できる．このような共進化環境では，色々なレベルのゲーマー能力を持ったプレーヤモデルが特定難易度のゲームになるようにゲームパラメータを仕上げることができるため，提案枠組みは色々な能力差のあるゲームプレーヤ向けにゲーム開発に有用である．

Contents

Chapter 1

Introduction

1.1 Background and Remaining Problems

Games have become a crucial part of human culture and one of the oldest forms of human social interactions. Common characteristics of games include mutually accepted rules and decisive goals, competition and cooperation, challenge and enjoyment as well as chances and unpredictable outcomes. They are used as pastimes, religious events, relationship-building activities, or teaching tools. Teaching functionalities of games range from skill development to ethical and strategic thinking lessons. Nowadays, people play games for various purposes and in numerous formats, particularly board games and video games.

Video games are now playing an important role in entertainment industry with the global revenues of US$79.7 billion in 2016 [11]. At 5.6% growth rate, the value of video game industry will have probably reached US$100-billion by 2020. Only in the United States, the world's biggest market for video games, the industry earned $16.8 billion in 2016; the Entertainment Software Association reported that more than 60% of its population play video games. In average, US gamers are spending 8.1 hours per week (12% of their leisure time) playing games and US$20 per month shopping on games [38]. Responding to the flourishing popularity of video game playing, either as a pastime or a serious hobby, thousands of new video games are commercially produced each year. In addition, hundreds of new games are available to the public for free.

As pastime games in portable devices are getting more and more popular, the number of game titles has been increasing and the production quality has been improved. To cope with such a huge demand, video game production must be developed faster with higher quality. In general, a blockbuster game for a serious hobbyist may take approximately 18 - 36 months for the production time, while a

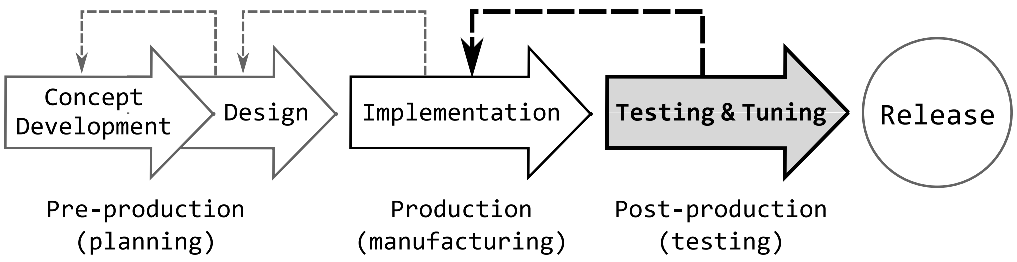

Concept Development
Design
Implementation
Testing & Tuning
Release
Pre-production (planning)
Production (manufacturing)
Post-production (testing)

Figure 1.1: Three major stages of game development process.

typical game for a typical player may take around 6 months or so [14].

To create a video game, the game development process comprises three major stages: planning, manufacturing, and testing [48] as illustrated in Fig. 1.1. In the planning stage, game developers initiate the concepts, work out the fundamental rules and mechanics, establish the primary designs, and set up all necessary resources for the game. After that the game contents, i.e. audio and visual elements, are created and integrated during the production phases. Finally, playtesting and game tuning are carried out to detect and resolve problems as well as to adjust the game to meet its required quality standards. The testing and tuning stage, normally, tend to delay the development of a video game due to the huge amount of work involved [18].

The main purpose of game testing and tuning is to make sure that the game will meet the required objectives when released. Usually, one of the most common objectives in video game development is to create an entertaining game which should neither be so difficult that a beginner is discouraged nor so easy that a skilled player gets bored [47]. This kind of balance in playing a game challenges the game players and engages their interest.

Many factors get involved in creating game engagement. The key point to entertain the players is to adjust the game parameters properly. Game parameters are variable settings in a video game which control the game difficulty. They are constants or variables in the source code so cannot be directly set by the players.

To achieve a game balance since the beginning of the game, game parameter tuning aims at determining the default values of game parameters while game developers are developing the game. This work is different from the dynamic game balancing [1], in which the game parameters are adjusted dynamically to catch up with the player's skills while a player is playing.

As it is not easy to manually tune game parameters to create an enjoyable and attractive game, the game designers must tweak them iteratively to achieve a balance of game difficulty [45]. This is a truly laborious and time-consuming task.

If we can reduce game developers' burdens and time wasted in a game testing and tuning process, the production time will become shorter. This means more chances for game developers to refine their games for better quality and deliver their products in time. Hence, extra tools and novel methodology to facilitate and boost up the testing and tuning processes are necessary.

1.2 Objectives and Approaches

Game parameter tuning is a crucial part of the game testing process. In a conventional procedure, the data gathered from testing are investigated and analyzed to improve the current version of the video game being tested. It is common to carefully choose human players, either professionals or novices, who can meet the objectives in verifying the games. This process, however, can be made automatic with simulated players who, when equipped with diverse gaming skills, will make automatic game parameter tuning feasible.

Our goal is to design a methodology to assist video game developers working on a game tuning process. The system should allow the developers to easily make use of their game expertise in the process. To waste less time in laboriously repetitive tunings, the system must perform automatically or semi-automatically with minimum efforts from the developers.

To accelerate the tuning process in a controllable way, we use the developers' game insights in guiding the tuning direction. Without prior knowledge, searching for suitable game parameters is likely to take longer time due to a vast search space. At the end, the results may be impractical for the real-world use. With the guiding information, fortunately, the tuning process tends to produce faster and more satisfying results.

Our approach uses computational intelligence (CI) techniques for knowledge-based modelling and optimization capabilities to solve the problems, basing on empirical models instead of analytical mathematical models. The methods can be verified by observation or experience, rather than theory or pure logic. This is suitable to our main objective to employ prior knowledge from game developers. Some CI techniques, such as fuzzy logic system (FS), require less technical skills for non-technical persons, e.g. game designers, to express their expertise in the modeling domain.

Although many artificial intelligence (AI) and CI applications are currently used to automate game content generations, those for automatic game parameter tuning are few [63]. Unlike other areas of automatic game generation which are conceptual and self-contained, the work of game parameter tuning is considered game-specific and the results are non-transferable.

Accordingly, we base our proposed methodology on a turn-based strategy (TBS) video game. In our scope, TBS game is a battle game where a human player takes turns in playing the game in a tactical duel with a computer-controlled player. The game is different from current popular real-time strategy games in which all players play the game simultaneously. Both kinds of games are originally derived from traditional board games, e.g. chess, go or checkers. They are ones of the most popular genres in the video game industry.

1.3 Chapter Structure

Following this introductory chapter, we present an overview of related CI techniques, as well as related research on AI & CI in games and automatic video game generation in section 2.1 and 2.2 of Chapter 2, respectively. The overview of our framework for automatic game parameter tuning is presented in Chapter 3. We then discuss the creation of our game player model designed for a TBS game in Chapter 4.

We improve the performance of the player model with CI-related optimization techniques. The implementation of an optimized player model is presented thoroughly in Chapter 5. Next, we examine the interaction between the optimized player model and manual game parameter adjustment in Chapter 6. In Chapter 7, we draw some discussions from the experiments in previous chapters to support the validity of our framework.

We wrap up our research in Chapter 8 for the conclusions and our future works. Fig. 1.2 shows the chapter structure of this dissertation graphically.

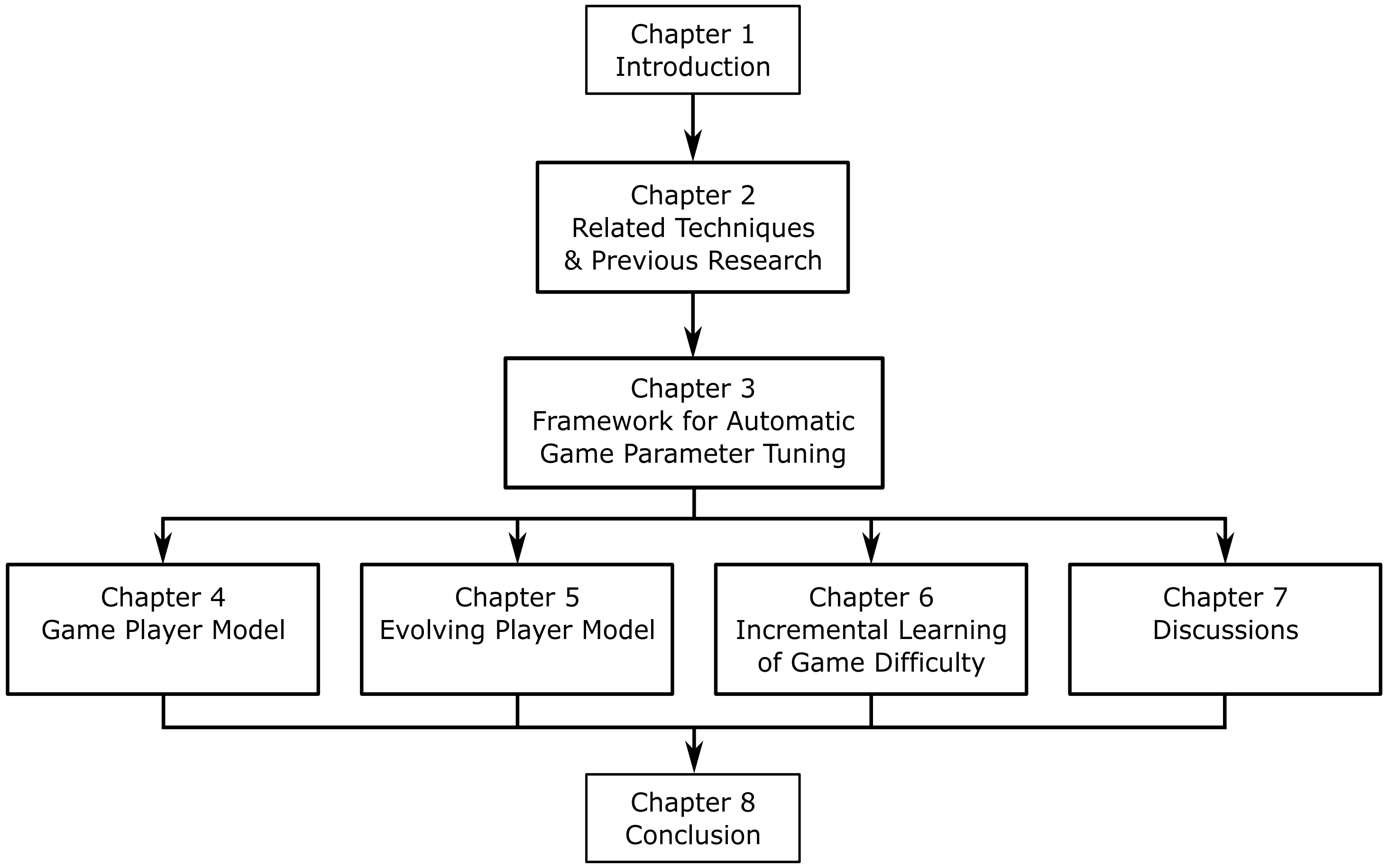


Figure 1.2: Chapter structure of this dissertation.

Chapter 2
Related Techniques and Research

2.1 Related Techniques

2.1.1 Computational Intelligence

Computational intelligence (CI) is a modeling of biological and natural intelligence created to solve complex problems. There are several main paradigms considered as CI. For example, artificial neural network (ANN), a model of a biological neuron analogous to signaling of the synapse in a human brain, searches for patterns or human perceptions. Inspired by the study of colonies in social organisms, swarm intelligence (SI) replicates social behavior of an individual organism in a swarm, looking for efficient collective solutions. Mimicking Charles Darwin's concept of *survival of the fittest*, evolutionary computation (EC) simulates natural evolution process to hunt for the best individual fitting our requirements. Fuzzy systems (FS) allows human reasoning with ambiguity and uncertainty to become expressible and computable.

Our proposed methodology for automatic game parameter tuning employs three major techniques from two main paradigms: Fuzzy rule-based system from FS paradigm as well as differential evolution (DE) and coevolution algorithm (CEA) from EC paradigm. In this section, we describe these three techniques used in our approach. In subsection 2.1.2, we begin with a fuzzy rule-based system which is the main component of our game player model. After that, we briefly summarize EC paradigm in subsection 2.1.3 before explaining DE and CEA algorithms in subsection 2.1.4 and 2.1.5, respectively.

2.1.2 Fuzzy Logic System

Fuzzy logic system (FS) is a rule-based system using fuzzy logic to express the relationship between causes and consequences. Lofti Zadeh proposed the concept of fuzzy logic and fuzzy sets in 1965 [65]. Since then, the idea has been furthered in many research fields and applications including knowledge-based system, pattern recognition, multi-objective optimization, evolutionary computation, etc. [55]

In this subsection, we present a fundamental concept of fuzzy logic and fuzzy sets. After that, we discuss the fuzzy IF-THEN rules and the fundamental mechanisms of fuzzy rule-based system. Finally, we examine some major concerns in the implementation of FS applications.

2.1.2.1 Fuzzy Sets

A fuzzy set is a set that its members contain a *degree of membership* (DOM). The DOM value is proportional to the likeliness of being a fuzzy set member. A *membership function* of a fuzzy set, μ_{fuzzy}, maps a given value x in the set U to the real interval from 0 to 1, as shown in Eq. 2.1.

$$\mu_{fuzzy}(x) : U \rightarrow [0,1] \tag{2.1}$$

A classical set, or a crisp set, is a subset of the fuzzy set. As shown in Eq. 2.2, the DOM value of a crisp set is either 1 or 0, i.e. existing or non-existing in the set. For any x in a crisp set S,

$$\mu_{crisp}(x) = \begin{cases} 1, & \text{if } x \in S \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

Membership functions of a fuzzy set may come in any arbitrary forms. However, they are normally in smooth and continuous shapes. The shape of a membership function defines the characteristics of a fuzzy set. Different views of vagueness in the set can be expressed through different shapes of its membership functions. Parameters that control the shape of a membership function are significant in an interpretation of the fuzzy set. Thus, we can control the characteristics of the fuzzy set via these parameters, which are often called *membership function parameters.*

2.1.2.2 Fuzzy Logic

Boolean logic, or two-valued logic $\{true, false\}$ or $\{0, 1\}$, is an operation associated with a crisp set. Both Boolean logics and crisp sets produce an exact reasoning system. They enable binary computation to work, which plays an important role in the development of a computer system.

Although Boolean sets and logics are very effective in a precise computation, many real-world problems are imperfect and ambiguous. This is emphasized using uncertain terminologies in human's languages. Equipped with a degree of certainty, fuzzy logic has became a crucial tool used to solve a lot of real-life problems and to model human reasoning.

Like crisp sets, fuzzy sets contain similar operations acting upon the sets, but with extra calculation on membership functions. The frequently used operations are union (OR), intersection (AND), and complement (NOT). For a particular operator, the resulting membership function is computed in different ways.

Union of fuzzy sets (OR): The union of two fuzzy sets contains all elements from both sets. However, the degree of membership depends on the specific union operator used. The example of commonly used union operators are:

- maximum: $\mu_{A \cup B}(x) = max\{\mu_A(x), \mu_B(x)\}$
- bounded summation: $\mu_{A \cup B}(x) = min\{1, \mu_A(x) + \mu_B(x)\}$

where U is a universe, A and B an arbitrary fuzzy set in U, x an arbitrary element of U, and μ_A and μ_B membership functions of A and B, respectively.

Intersection of fuzzy sets (AND): The intersection of two fuzzy sets contains any elements existing mutually in both sets. However, the degree of membership depends on a specific intersection operator used. The example of commonly used intersection operators are:

- minimum: $\mu_{A \cap B}(x) = min\{\mu_A(x), \mu_B(x)\}$
- product: $\mu_{A \cap B}(x) = \mu_A(x) \cdot \mu_B(x)$
- bounded product: $\mu_{A \cap B}(x) = max\{0, \mu_A(x) + \mu_B(x) - 1\}$

Complement of fuzzy sets (NOT): The complement of fuzzy sets contains the same elements, however, with the inverse degree of membership. This is a generalization of the complement operation in a crisp set.

- complement: $\mu_{\neg A}(x) = 1 - \mu_A(x)$

2.1.2.3 Linguistic Values and Linguistic Variables

We use a fuzzy set to represent a natural language term describing a quantitative or qualitative concept, for instance "near", "far", "large", "warm", "rich", "strong", etc. The term is often called a *linguistic value*, a *fuzzy value*, or a *fuzzy label.* A group of linguistic values forms a *linguistic variable* or a *fuzzy variable* to express the reasoning or information with imprecise or imperfect characteristics. Illustrated in Fig. 2.1, a linguistic variable “walking distance” contains three linguistic values “near”, “average”, “far” to express different levels of a distance in a human language. In fuzzy logic applications, the linguistic values are often used to facilitate the expression of facts and rules. A fuzzy variable is a basic building block of a fuzzy rule.

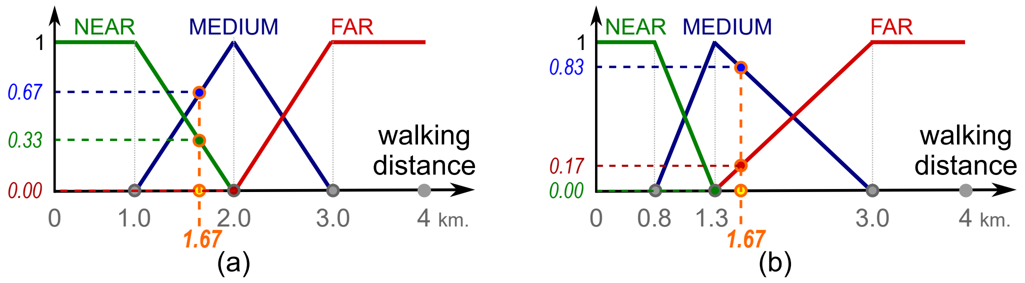


Figure 2.1: A linguistic variable *walking distance* contains three linguistic values *near*, *medium*, and *far*. For the same linguistic value, different forms of membership functions provide different interpretations and fuzzification values. At the distance of 1.67 km., figure (a) has the membership values $\mu_{near}(1.67) = 0.33$, $\mu_{medium}(1.67) = 0.67$, and $\mu_{far}(1.67) = 0.00$, while figure (b) has $\mu_{near}(1.67) = 0.00$, $\mu_{medium}(1.67) = 0.83$, and $\mu_{far}(1.67) = 0.17$.

The expression “a station is near” forms a fuzzy statement in which we can determine a degree of membership. We use the membership function associated with “near” label in “walking distance” fuzzy variable to determine the quantitative value of the closeness to the station. The process of mapping a given input value (actual distance to the station) to its corresponding degree of membership in a fuzzy label (quantitative value of the closeness to the station) is called *fuzzification*. Shown in Fig. 2.1, applying fuzzification process with different shapes of the membership function results to different degrees of the membership.

2.1.2.4 Fuzzy IF-THEN rules (Conditional fuzzy rules)

Fuzzy IF-THEN rules are used to express human knowledge very effectively. The main advantage is that it allows linguistic reasoning with high interpretability, an ability to understand the meaning. It is easy not only for an expert to create fuzzy if-then rules from his or her skills in the domain, but also for an ordinary person to understand the underlying wisdom in the field.

The fuzzy if-then rule is in a form of:

$$\text{IF } antecedence(s) \text{ THEN } consequence(s).$$

or

$$\text{IF } \textit{condition term}(s) \text{ THEN } \textit{result term}(s).$$

Antecedence is a causal statement or a combination of several causal statements establishing a given condition. It defines the area of interest in the problem domain. Consequence, on the other hand, is a result or a conclusion inferred from the given antecedence(s). This is a very simple form to represent fuzzy rules. This section will describe the fuzzy if-then rules along with a concrete example. It will also explain all the processes to obtain the output result from the given rules.

Fuzzy Condition Statement (Antecedence): We can combine several single fuzzy statements all together with fuzzy logic operations (AND, OR, NOT) to create a compound fuzzy statement. This kind of statements is used as a causal statement to describe an input condition in a fuzzy if-then rule structure. It is called *antecedence*, or the IF-part of a rule. For example, a compound statement "IF a station is NOT near AND money is large" applies NOT operation to the first statement and combines two single statements with AND operation. To compute the numerical value for this compound statement, we first perform fuzzification on each single statement and apply fuzzy operators on each term afterward. This calculation process of IF-part statements is called antecedent activation.

Fuzzy Result Statement (Consequence): The THEN part of a fuzzy if-then rule is called *consequence*. It indicates the result of the input conditions (IF-part statements). The process to calculate the resulting influence in each rule is called *implication*. As the conclusion of each rule is drawn upon its inputs, we first determine the rule's weight (or truth value) from antecedent activation process. Then, use this weight to imply the consequence in the rule according to an implication operator: minimum, product, etc.

The consequence can be defined in many forms that categorize the types of fuzzy systems. Two major models are:

1. **Mamdani model** has fuzzy sets as a rule consequence. Mamdani proposed this model in 1974 as fuzzy logic controller [34]. It is originally used to translate human experience into controlling rules. The output consequences of this model are linguistically defined with meaningful terms, leading to its high

interpretability. The rigidity of the linguistic values, however, limits the system accuracy. Moreover, the computation to obtain the final results from fuzzy outputs is costly. Thus, another model to overcome this problem is later introduced.

2. **Takagi-Sugeno-Kang (TSK) model** has a function of input variables as a rule consequence, instead of introducing new fuzzy output variables. The output is now easier and more accurate to calculate, with the given function. However, it is more difficult to understand the relationship between IF-clauses and THEN-consequence function. This model is proposed in 1985 by Takagi and Sugeno [56] then refined in 1988 by Sugeno and Kang [53].

There are several hybrid models constructed from both Mamdani and TSK models. One of the hybrid models worth mentioning is a singleton fuzzy system. This model has a constant as a rule consequence. This constant is considered as a one-value crisp set, i.e. a subset of fuzzy set, in Mamdani model or a constant value of an output function in TSK model. The singleton model offers a good balance between Mamdani and TSK model: fast computation with high interpretability.

Crisp Output: After applying implication process to all rules, we obtain several fuzzy set outputs for each rule. To achieve a single decision, we then combine these outputs into a single fuzzy output. This process is called *aggregation*. Two commonly used aggregation operators are maximum and bounded summation methods for union operators (see union operation in subsection 2.1.2.2).

Finally, the fuzzy output from aggregation process should be converted into a numerical value for the decisive result. This is an inversion of a fuzzification process. Hence, we call the task as *defuzzification*. There are several defuzzification methods. The most commonly used methods are *center of area* and *mean of maxima* [40].

2.1.2.5 Fuzzy rule-based system

A rule-based system uses some rules to describe the mechanism of an event or a decision. Similarly, a fuzzy rule-based system uses the concept of fuzzy sets and fuzzy logic to create the rules to describe the mechanism. The fuzzy rule defines the causal relationship between linguistic input variables representing fuzzy sets and its corresponding outcome(s).

The basic elements of fuzzy rule-based system comprise of 4 main components as illustrated in Fig. 2.2.

1. A knowledge base (fuzzy rules + membership functions)
2. A fuzzifier converts crisp inputs into fuzzy values.
3. A reasoning engine calculate a fuzzy output according to fuzzy reasoning processes.
4. A defuzzifier convert a fuzzy output back into a numerical result.

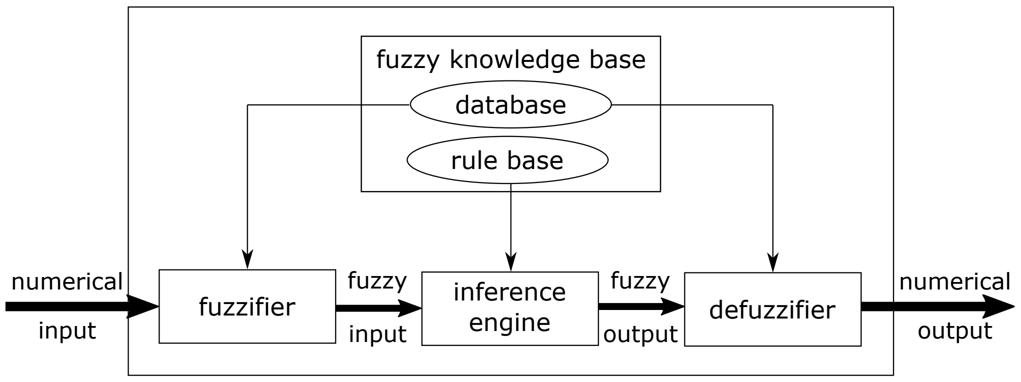


Figure 2.2: A fuzzy rule-based system.

2.1.2.6 FS Implementation Concerns

The main concern of the FS implementations is the lack of scalability. The number of fuzzy rules depends largely on the number of linguistic input variables and the number of their corresponding linguistic labels. Suppose we have three linguistic variables and each variable contains three linguistic labels. It requires $3^3 = 27$ fuzzy rules to specify all relationship among them. When a new variable of three linguistic labels is added to the system, we need to specify $3^4 = 81$ rules to represent all associations. Hence, the rule size grows exponentially. We can say that the fuzzy rule-based system is suitable for a small number of input variables and a small number of linguistic labels in each variable.

With Combs method [7], however, we can break down a single FS rule with a number of ANDed fuzzy variables into several FS rules with a single FS variable. That is:

$$\text{IF a AND b THEN c} \equiv \text{(IF a THEN c) or (IF b THEN c)}$$

where a and b are fuzzy statements. By limiting the rule to the Combs format, we can create fuzzy rules of which their size can be scaled linearly on the number of input states. However, the Combs method holds its validity as long as the decomposed rules are all consistent [35].

To create efficient fuzzy rule-based systems, we depend essentially on the expertise of the professionals. Human experts not only design the fuzzy rules from his experiences, but also supply the relating membership functions to interpret linguistic input variables for fuzzy reasoning.

An important thing to consider in creating fuzzy rules is choosing consequence models: Mamdani, TSK, or hybrid model. This decision concerns the trade-offs among rule interpretability, accuracy, and speed of the computation.

Creating membership functions is not an easy job for human experts. Very fine details of membership functions are required to get most optimal results. We usually get a good result from a set of human-supplied membership functions, but not always the best one. The result may not be fully optimized to achieve the best result due to the limited capability of human beings.

In summary, fuzzy rule-based systems provide an efficient way to allow human experts to produce the governing rules in a natural way using verbal expressions. The system transforms qualitative statements into quantitative values for precise computation. Unlike other black-box methods, e.g. neural networks and their variants, fuzzy rule-based systems show the logic behind the thinking way in the controlling rules. With such high interpretability, it is easy to create, understand, and maintain the system for both rule developers and users.

2.1.3 Evolutionary Computation

2.1.3.1 Biological Evolution & EC Algorithms

In biology terminology, *evolution* represents changes in the heritable characteristics of biological populations over successive generations. In 1859, Charles Darwin published *On the Origin of Species* with the theory of evolution by natural selection. The major element of natural selection is variation. Natural variation refers to differences in genes of each individual population. The process of permanent alteration in genes is called *mutation*. This also leads to diversity in physical characteristics. Some variations may improve an individual's chance of survival from threats in competitions with enemies and environments. It is as if the dangers eliminate the unfitted and keep only the well-adapted individuals. The process of survival from fatal threats is called *selection*. The survivals, with special characteristics, tend to create more offspring due to their longer lifetime. They transmit the special characteristics to their offspring through the genetic inheritance. The process to create new offspring by combining the parents' genes is called *crossover*. The repetitive adaptation of genes to survive over several generations creates the survival of the fittest.

An EC algorithm adapts the above three main processes in biological evolution, i.e. crossover, mutation, and selection, to solve global optimization problems. The algorithm contains a population of individuals. Each individual consists of a sequence of genes representing a solution to the problem. The algorithm applies biological evolution processes to the population iteratively. Each iteration tries to improve the population and passes down the fittest sets of genes from one generation to the next. An *EC generation* represents an evolving population at a specific iteration during the evolution process. An EC algorithm is categorized as a population-based, trial-and-error, stochastic problem solver.

To solve a problem with an EC algorithm, we randomly generate a set of candidate solutions as an initial generation. Next, for each successive generation, we improve its population by removing less-desired individual solutions (aka selection) and reproducing new solutions from the decent individuals (aka crossover). In addition, we also modify the new individuals by applying small changes stochastically to search for a better solution (aka mutation). We apply these three processes repeatedly until reaching the target solutions or computation limits. The method offers a metaheuristic approach to solve incomplete or imperfect optimization problems.

2.1.3.2 EC Implementation Concerns

In addition to three main natural evolution processes, there are several issues related to the implementation of EC algorithms. For example,

- **An encoding chromosome to solutions:** The encoding maps a problem representation into a parameter space in which we search for solutions. A good encoding must represent the problem clearly and comprehensively in order to make a search easily.

- **An initialization of initial solutions:** We begin with defining the boundary constraints of an encoding parameter space. After that, we initialize an initial population to cover the parameter space uniformly. With some knowledge in the problem domain, we may initialize the initial population specifically to accelerate a search process.

- **A fitness evaluation function:** We need a way to evaluate the fitness or the survival strength of an individual. This function is often called an *evaluation function* or a *fitness function*. It maps the quality of an individual solution to a real number called *fitness value*. We use the fitness value to separate good solutions from the bad ones. The evaluation function represents selection by the nature.

- **Search exploration and exploitation:** On the one hand, search exploration is the process that widely examines unknown regions in the search space looking for a new prospect solution. On the other hand, search exploitation is the process that locally investigates a previously visited region to refine the search results in the highly-anticipated area. To achieve a good search, we need to balance between exploration and exploitation. A general guideline is to focus on exploration in the earlier search and exploitation in the later search. Each of EC algorithms differs in its components contributing to exploration and exploitation as well as the issue of when and how exploration and exploitation are controlled [8].

2.1.4 Differential Evolution

2.1.4.1 Overview of DE Algorithm

A Differential Evolution (DE) is a stochastic, population-based search and an optimization technique inspired by an EC. Proposed by Rainer Storn and Kenneth Price in 1997 [52], a DE algorithm is originally used for multi-dimensional, continuous, real-value problems. The recent developments of DE have been applied to discrete-valued, multi-objective, and multi-modal problems [9], [10].

The main difference between a DE and other EC algorithms is the use of direction and distance information in the current population to guide the search process. We obtain the direction and distance information in a form of a difference vector, which hints the name of this algorithm.

2.1.4.2 Generic DE Algorithm

In a common DE implementation, we start with an existing individual in a population. We refer to this individual as a *target vector.* A DE algorithm randomly picks other individuals from the population and creates a difference vector. The *difference vector* is a vector that originates from one random individual toward another random individual. Next, the DE applies mutation process upon the difference vector and a *base vector*, an individual in the population chosen by a DE control policy. The mutation process creates a *mutation vector.* Next, the DE applies crossover process between the mutation vector and the target vector. The crossover process creates a *trial vector* which is a new location to be searched for a solution. Then, we evaluate the fitness value of the trial and target vectors. Subsequently, the DE applies selection process to preserve the better vector between the trial and target vectors for the next generation.

We apply these procedures to all individuals in the current generation before advancing to the next generation. The algorithm iterates the same procedures until satisfying one of the following stop conditions: (i) the maximum number of generations, or (ii) the maximum count of fitness evaluations, or (iii) the specific fitness value of the best individual.

Algorithm 1 shows a generic DE algorithm. The algorithm begins with the initialization of its parameters between line 1 - 3. Then, the algorithm enters a while-loop in line 4 to perform its searches for each generation of a DE population. Here, the algorithm terminates if one of the stop conditions is met. After that, from line 6 - 24, the algorithm iterates over each individual in the population and applies mutation, crossover, and selction processes to the individual starting at line 6, 11, and 19 respectively.

2.1.4.3 DE Mutation Process

The main objective of the DE mutation process is to generate a mutation vector. We illustrate the mutation process in Fig. 2.3. First, a difference vector is created from a difference between two random individuals, x^{r_2} and x^{r_3}. The difference vector is rescaled by a control parameter, named *a scaling factor F*. The scaling factor is a real value and is suggested to be between zero and two [52]. We add the difference vector to another random individual x^{r_1}, called a base vector, to create the mutation vector.

Algorithm 1 Generic differential evolution (DE) algorithm.

```
 1: Initialize generation counter: $g = 0$.                                  ▷ initialization
 2: Initialize scaling factor $F$ and crossover rate $CR$.
 3: Randomly Initialize a population $P$ of size $NP$ with $D$-elements individuals.
 4: while (stop condition is not true) do

 5:     for i=1 to $NP$ do
 6:         Pick base vector $x_{base}$ from $P$.                                 ▷ mutation
 7:         Pick random vectors $x_{r_1}, x_{r_2}$ from $P$ where $r_1 \neq r_2 \neq i$.
 8:         Calculate difference vector: $x_{diff} = x_{r_2} - x_{r_1}$.
 9:         Calculate mutation vector: $v_i = x_{base} + F \times x_{diff}$.

10:         Let $x_i$ denotes target vector and $u_i$ trial vector.               ▷ crossover
11:         Calculate random crossover point: $j_{rand} = rand(0,1) * D$
12:         for j=1 to $D$ do
13:             if $(rand(0,1) \leq CR)$ or $(j == j_{rand})$ then
14:                 $u_{i,j} = v_{i,j}$
15:             else
16:                 $u_{i,j} = x_{i,j}$
17:             end
18:         end

19:         Evaluate fitness score of target vector $f(x_i)$.                     ▷ selection
20:         Evaluate fitness score of trial vector $f(u_i)$.
21:         if $(f(u_i)$ is better than $f(x_i))$ then
22:             Store trial vector for the next generation: $P_i = u_i$.
23:         end
24:     end

25:     Advance to the next generation: $g = g + 1$.                        ▷ next generation
26: end
```

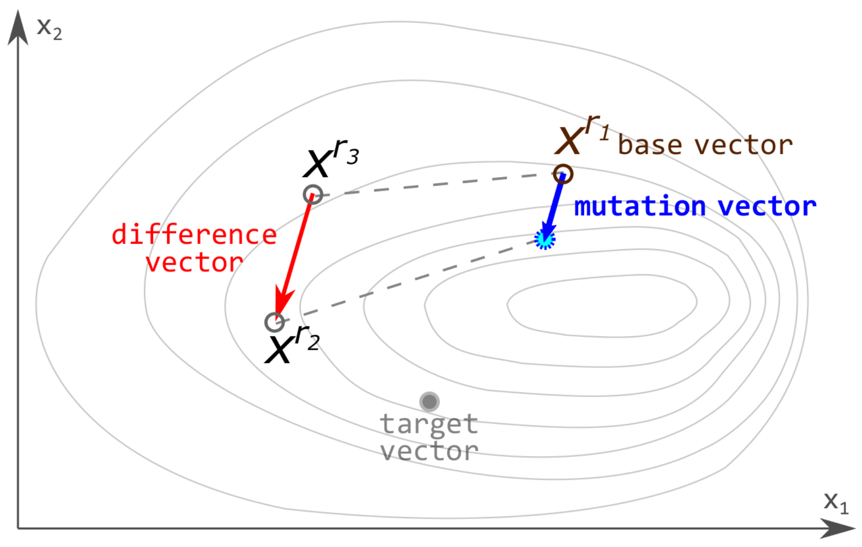


Figure 2.3: DE Mutation process generates a mutation vector, which is originated from the base vector. Its direction is proportional to the direction of the difference vector.

Equation(2.3) shows how to calculate a mutation vector, where v denotes a mutation vector, F a scaling factor, and $x_{r_1}, x_{r_2}, x_{r_3}$ three random individuals from the population.

$$v = x_{r_1} + F \times (x_{r_2} - x_{r_3}) \tag{2.3}$$

2.1.4.4 DE Crossover Process

The main objective of the DE crossover process is to generate a trial vector. To help increase diversity of a searching location, we crossover the mutation vector with the target vector to form the trial vector. Another control parameter, named *a crossover rate* CR in Eq.(2.4), is introduced here as a threshold to randomly choose the element values between the two vectors. This operation finally generates the trial vector, a new searching location. Figure 2.4 shows two possibilities of the trial vector generation in a two-dimensional search space.

Equation(2.4) shows the crossover process; where x_j, v_j, u_j denote the j^{th} element of the target, mutation, and trial vector, respectively, and j_{rand} is a random element in the mutation vector.

$$u_j = \begin{cases} v_j, & \text{if } rand_j(0,1) \leq CR \text{ or } j = j_{rand} \\ x_j, & \text{otherwise} \end{cases} \tag{2.4}$$

The second conditional term of $j = j_{rand}$ guarantees that at least one element of

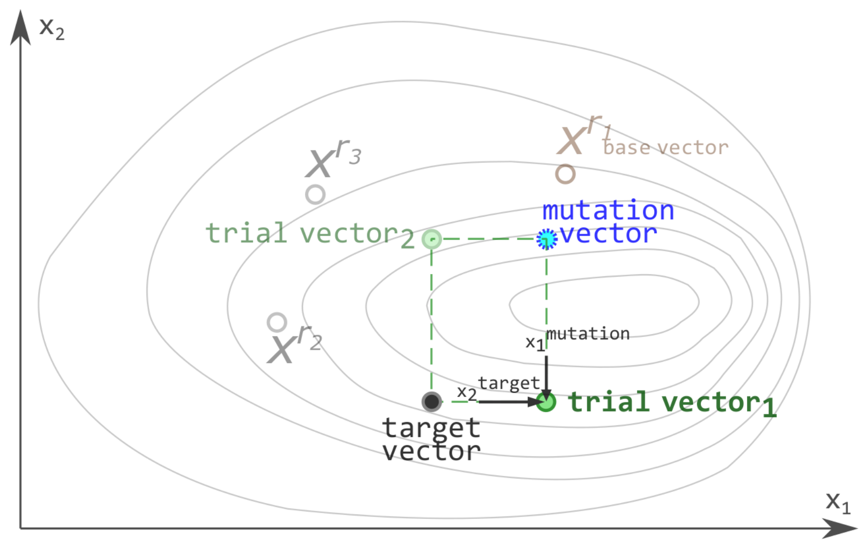


Figure 2.4: DE crossover process generates a trial vector from two parent vectors, i.e. the mutation and the target vectors.

the mutation vector is swapped to create the trial vector. This is to make sure that, despite a low crossover rate CR, the trial vector is always different from the target vector.

2.1.4.5 DE Selection Process

The main objective of the DE selection process is to determine a better individual for the next generation, between a descendent and a parent vectors. The process first evaluates the trial and the target vectors, then compares their evaluations, and keeps the one with the better fitness score. Figure 2.5 shows the selection process between the trial and the target vectors.

2.1.4.6 Variations of DE Algorithm

Variants of DE algorithms differ in the detail of mutation, crossover, and selection implementations. We can modify the mutation operation, as shown in lines 6 - 9 of Algorithm 1, to control the exploration and exploitation behaviors of the DE algorithm. By changing the way to create a mutation vector in Eq. 2.3, specifically how to pick a base vector and how to create a difference vector, many behaviors suitable for searching different types of fitness landscape can be achieved.

The notation of basic DE strategy, as explained in subsection 2.1.4.3, is called *DE/rand/1*. The notation scheme in Table 2.1 is in the form of *DE/x/y* where x represents how to pick a base vector, and y represents how many difference vectors are used. The possible values of x include:

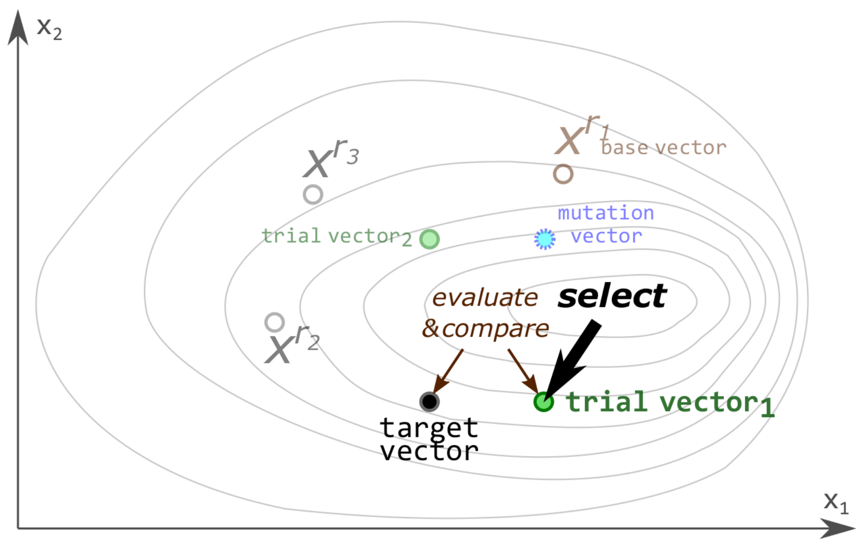


Figure 2.5: DE selection process selects the better between the trial and the target vectors, according to its fitness evaluations.

- *rand* indicates a randomly selected base vector.
- *best* indicates the best individual of the population as a base vector.
- *target-to-best* indicates a base vector that originates from a target vector and points toward the best individual of the population.

The table shows some variants of DE algorithms along with their corresponding mutation equations and brief comments.

2.1.4.7 Control Parameters in DE Algorithm

According to Algorithm 1, the major control parameters are

- The population size NP from initialization in line 3,
- The scaling factor F from mutation process in line 9, and
- The crossover rate CR from crossover process in line 13

Many adaptive DE algorithms adjust these three parameters, in a specific combination, to accelerate the evolution results [44].

Creating a trial vector from a difference vector is a good strategy to automatically balance between the exploration and exploitation searches. In the initial generation of a DE algorithm, individuals are randomly initialized all over the search space. As

Table 2.1: Variations of DE algorithms. The second column shows a variant of Eq.(2.3) where v denotes a mutation vector, x a target vector, $x_{r_1}, x_{r_2}, x_{r_3}, x_{r_4}, x_{r_5}$ random individuals different from x, x_{best} the best individual in a population, and F a scaling factor.

Notation scheme	Variants of mutation equation Eq.(2.3)	remarks
DE/rand/1	$v = x_{r_1} + F \times (x_{r_2} - x_{r_3}))$	A random individual encourages exploration behavior.
DE/rand/2	$v = x_{r_1} + F \times (x_{r_2} - x_{r_3}) + F \times (x_{r_4} - x_{r_5})$	Two difference vectors are generated from four random individuals.
DE/best/1	$v = x_{best} + F \times (x_{r_1} - x_{r_2})$	The best individual encourages exploitation behavior.
DE/target-to-best/1	$v = x + F \times (x_{best} - x) + F \times (x_{r_1} - x_{r_2})$	A target-to-best vector balances both exploration and exploitation.

a result, the algorithm creates difference vectors with large magnitudes due to the scattering of its population. This allows trial vectors to spread out and explore the search space during the early generations. After the population evolves over times, individuals will converge to a potential optimal solution. In the later generations, the magnitude of difference vectors becomes smaller than the earlier generations due to the gathering of the DE population. The trial vectors gradually change from exploring a wide area to examining a local region near the optimal solution. Hence, the exploration automatically transforms into the exploitation in an accordance with the population distribution in the search space.

2.1.5 Coevolutionary Algorithm

A Coevolutionary Algorithm (CEA) is a natural extension of an EC where each individual is evaluated based on its interactions with other individuals. For standard EC algorithms, an individual is evolved among its population in a fixed environment. In contrary, CEA proposes that the environment is simultaneously influenced by all populations which interact with one another independently. Hence, each population, in addition to local adaptation within its own population, takes turn to evolve in response to the ever-changing environment, caused by the evolving populations.

Unlike standard EC algorithms in which each individual is evaluated according to an explicit fitness function, a CEA uses the performance comparison between populations as an evaluation measure. The performance comparison measures the interaction success of an individual in relation to other individuals of the same population. This interaction measure is also called a relative fitness. With a CEA, we can search for the solutions without prior knowledge of the problem domain or an explicit fitness function of the population. Thanks to the interactions, the relative performance comparison drives the coevolution process between interacting populations [15].

One of the significant features of a CEA is to allow the individuals to escape from stagnation in local optima. When the environment is gradually altering over time, the algorithm perturbs the fitness landscape in such a way that some individuals are able to migrate from the local optima [5].

In a CEA, different populations normally play different roles. We categorize a CEA into two major groups: (1) competitive CEA and (2) cooperative CEA. On the one hand, a competitive CEA establishes an *arm race condition* where an advantage of a population implies a disadvantage of the other. Each population focuses on outperforming the other. On the other hand, a cooperative CEA set ups a *symbiotic condition* where all populations mutually benefit from their collaboration. Both groups of CEAs contain different concepts of population interactions as well as controlling policies. In the following subsections, we describe the generic algorithm of coevolution, then explain the characteristics of both competitive and cooperative CEAs. We focus on a competitive CEA because it is the basic mechanism for our proposed methodology.

2.1.5.1 General CEA Algorithm

The algorithm 2 shows the idea of how coevolution works in general. For each generation, we first calculate the relative fitness of each individual in every population group and then evolve each population group afterward. We can use any EC algorithms to evolve each population group distinctively. To calculate relative fitness of an individual, we select some individuals from the opponent groups as the competitors/collaborators and use the competition/collaboration results to compute the relative fitness.

Algorithm 2 Generic Coevolutionary Algorithm

```
1: Initialize generation counter: $g = 0$.
2: Initialize all populations $\mathbf{P}$.
3: do
4:     foreach population $P_i \in \mathbf{P}$ do                ▷ calculate relative fitness
5:         Sample opponent individuals $S$ from each populations $\{\mathbf{P} - P_i\}$.
6:         foreach invidiual $C_i \in P_i$ do
7:             Evaluate relative fitness of $C_i$ with respect to sampling populations $S$
8:         end
9:     end
10:    foreach population $P_i \in \mathbf{P}$ do                ▷ evolve using EC
11:        Evolve population $P_i$ to the next generation.
12:    end
13:    Advance to the next generation: $g = g + 1$.
14: while (stop condition is not true)
15: Select the best individual as the solution.
```

2.1.5.2 Competitive Coevolution

In this algorithm, we reward an individual when it works well against the other populations. In this case, each population tries to compete with one another and become better and better at its defensive and offensive strategies in each generation.

This is similar to a predator-prey model in biological systems, where a predator keeps developing its attacking tactics while its prey keeps improving its escaping strategies. Another analogy to the method is a host-parasite model, where a host represents a solution to solve a problem with data sets defined by a parasite. A solution host evolves to solve as many problems as possible, which, in turns, become more and more difficult due to the adapting data sets [21].

Controlling Policies: In a competitive CEA, there are several controlling policies to accelerate the coevolution. Two major concerns are of interest: (1) how to evaluate a relative fitness of an individual in one population from the performance comparison between several population groups, and (2) how to sample individuals from one population to coevolve with the others.

Relative Fitness Evaluating Policies: A CEA supports interactions between different types of populations. The fitness evaluation is a relative measure of each individual's performance in the whole interacting environment. We can measure this fitness in several ways. Below is the list of commonly used policies

[15]:

1. Simple fitness: For a given individual, we directly count the number of defeated opponent individuals, as a relative fitness score. In this case, the most winning individual is the fittest individual in the population.

2. Fitness sharing: For a given individual, we calculate a relative fitness score from the simple fitness divided by the sum of its similarities in the winning list. The most winning individuals may win over the same opponents similar to others, and they all share the same amount of fitness rewards. In this case, we oppose the similarity of the winning list and reward unusual individuals containing the uniquely maximum winning list.

3. Competitive fitness sharing: In this policy, we first compute the competitive fitness from the viewpoint of opponent individuals. For each opponent individual, the competitive fitness is inverse proportionally to the number of losses. The most losing opponent has the lowest competitive fitness, while the least losing opponent (aka the most winning opponent) has the highest competitive fitness. Hence, for a given individual, a relative fitness score represents a summation of the competitive fitness from the opponents it can win. This policy rewards the wins over the opponent that only a few other individuals could beat.

These policies are specifically designed for the competitive CEA, where a loss in one population group is a gain in the other group(s). This is contrary to the nature of the cooperative CEA, where a collaboration is necessary and different measures of relative fitness are required.

In addition to the fitness ranking for the selection process, we also use the relative fitness in the sampling policies to help reduce the computational expense of coevolution process.

Sampling Policy: This is a policy to match an individual from one population group to coevolve with individual(s) from different groups. The followings are the common sampling policies normally used in a CEA [15].

1. No sampling: This is one of the simplest forms of the sampling policy. We match all individuals in a population group to all individuals in the other, in

a round-robin fashion. It is the most time-consuming policy among all, yet it always gives the satisfying results because all possibilities are paired. This policy is good with a small-size population.

2. Random sampling: In this policy, we select a sample uniformly at random. Thus, the coevolution processes are computationally minimized, yet the quality of the coevolution is somewhat questionable.

3. Fittest sampling: To reduce the amount of coevolution while still preserving the quality results, we select only the fittest individual from the other population groups for the samples.

4. Tournament sampling: A tournament is a series of pair-wise matches where the winner advances to the next round, until the most winning individual is decided. We emulate a tournament of random matches to determine a sample.

5. Shared sampling: In this method, we select opponent individual(s) with most wins over the opposing population groups as the samples. (see competitive fitness sharing policy in the previous subsection.)

In addition to the competitive CEA, we can apply this sampling policies to the cooperative CEA. The explanation for the cooperative CEA is given in the next subsection.

2.1.5.3 Cooperative Coevolution

We reward an individual when it works well along with the other populations and benefits the whole environment. Meanwhile, we punish an individual when it works poorly with the others and spoils the environment.

Potter and De Jong proposed the cooperation approach of CEA in 1994 [42]. In their approach, each population group represents a component of a solution, and a collection of the best individual from these groups establishes a complete solution to the problem. The cooperative CEA coevolves independent components more efficiently than when the traditional EC evolves an entire solution. Algorithm 2 also shows the main idea of the cooperative coevolution algorithm.

As cooperative CEA focuses on the collaboration between subcomponents, the relative fitness evaluation should measure the effort contributed by an individual collaborator appropriately. We call this measurement a *credit assignment policy.* There are many policies to faithfully evaluate an individual subcomponent corresponding to its contribution. Three of the most often used policies are *optimistic, hedge,* or *pessimistic*; in which we assign an individual the value of its best collaboration, average collaborations, or worst collaboration, respectively [61].

2.1.5.4 CEA Implementation Concerns

Although the interactions among the population groups, both competitively and cooperatively, may help the evolution process refrain from local stagnation, there are times when the evolving process faces new dilemmas. Two of the most frequently found CEA dilemmas are *losses of gradients* and *cycling population dynamics* [4].

Losses of gradients occurs when a population group reaches a state that makes other groups lose fitness pressure to keep them from improving. For competitive CEAs, imagine when one population group is much more superior to its component group. In this case, the opponent group always loses and learns nothing from the competition. For cooperative CEAs, if a population group somehow loses its diversity instantly, the search space of its collaborator groups is suddenly limited in response.

Cycling population dynamics indicates a situation similar to the non-dominant, cyclical wins in the game of *rock, paper, scissors*, where a population group evolves back to the same state after several generations. There are several techniques to solve this problem. The concept of *hall of fame* where the fittest individuals from the previous generations are also included in the competitions is one of the well-known practical solutions.

2.2 Related Research

In video game development, several intelligence algorithms and techniques can be used to assist or automate the processes. In this section, we summarize previous researches relating to the automatic fine-tuning of video game parameters. We begin with a general survey of computational intelligence (CI) and artificial intelligence (AI) techniques in generic games. After that, we conduct a survey regarding automated video game generation. Finally, the findings of the surveys regarding three automation processes in game rules and mechanics, game content, and game tuning are presented, respectively.

2.2.1 Artificial Intelligence and Computational Intelligence in Games

AI and CI algorithms and techniques were proposed for several video game applications. Both techniques share the same goal in searching for machine intelligence, a study of intelligent machines that work and react like humans. However, they are different in the way they approach the problems. On the one hand, AI tries to simulate intelligence that can be programmed effectively, but, on the other hand, CI looks for a way that enables intelligence to emerge via statistical processes, usually driven by data or experience [32]. The two ways of thinking complement each other, providing different advantages and disadvantages to tackle the same problem. Nonetheless, this description is one of many explanations for the techniques and other opinions does exist. The main purpose to describe them here is to point out the main differences between the two commonly-used techniques in machine intelligence for games. In this dissertation, we use the term *machine intelligence* when refering to both AI and CI techniques in general. The mainstream usage of machine intelligence for games is to construct game controllers, or game agents, that can play a game well, without cheating the existing game rules.

2.2.1.1 Traditional Board Games and Machine Intelligence

Creating the automatic game controllers is one of the first efforts to implement machine intelligence techniques. Historically, the attempts focused on a machine playing a board game with a human being. Being widely recognized as a human-intelligence representation, chess playing showed us many famous, infamous and classic examples of the challenge. The controversial one is an automaton chess player called the Mechanical Turk, which is originally dated back to 1770. The machine turned out being a hoax with a human player hidden inside [16]! The evidence of solving chess with a general-purpose computer is commonly credited to Alan Turing, who suggested and commented on chess-playing as an opening challenge for a new-era thinking machine [58]. Claude Shannon, later, took the idea into practice and tackled the problem with a computer program. Since then, chess playing has become widely used as a research target for machine intelligence. In 1970, Association for Computing Machinery (ACM) held the first computer chess championship. This human vs machine competition in chess reached its peak on 11

May 1997, when Garry Kasparov, a World Chess Champion, closely lost to IBM's Deep Blue machine. Deep Blue is a supercomputer specifically built for the chess competition using a brute-force approach [16]. The machine is highly optimized for minimax-based algorithms and effective pruning techniques.

While sophisticated machines were outperforming human chess players, an interesting case occurred in 2005 Freestyle Chess Tournament, which welcomed all combined teams of human beings or computing machines. Two amateur chess players with three laptops won the tournament beating teams of grandmasters and supercomputers, from the qualifying rounds to the finals. To guide their moves, they used the laptops to run the self-developed chess-analysis program equipped with commercial chess software and custom-built database. This remarkable event shows that, as Garry Kasparov pointed out [25], "Human strategic guidance combined with computer tactical acuity was overwhelming."

2.2.1.2 Minimax Algorithm

Machines now defeat human players in all popular traditional board games. The common characteristics among these board games are, generally, alternate playing between multiple players and fully observable information on a game board. The rules of these game specify valid moves and well-founded interactions among the game objects. The combinations of all these moves and interactions produce gigantic game trees. A game tree is a directed graph in which each node represents game object's positions on the board and each edge represents a game object's move. The game tree is used to represent a whole game play, including all moves and interactions. To portray an actual game competition, we use a game tree with alternate moves that maximize one player's benefits and minimize the other's, consecutively until reaching an endgame or a certain tree depth. Then, for our game decision, we evaluate a positional cost for each decision path and determine the best one. This method of decision making, which takes turns to minimize and maximize the profits, is called *minimax algorithm.* Minimax is a fundamental algorithm for a turn-based game with discrete state. It is often referred to as a brute-force approach, following its intensive search of all possible decisions without learning behaviors.

When used in more complex games with excessive possibilities of valid moves, a minimax search could be very expensive since all paths in every tree branches must be evaluated. Thus, we need some heuristic methods to quickly cut down under-performed tree branches, so called a *pruning technique*. The idea behind the pruning is that, when a move on a branch is found worse than the previously evaluated one, we stop searching the branch. The most popular technique to reduce the tree search is an *alpha-beta pruning*. Alpha refers to the minimum scores from the maximizing player, while beta refers to the maximum scores from the minimizing player. Initially, alpha is a negative infinity, and beta is a positive infinity. When both values cross, meaning alpha becomes greater than beta, we stop searching the branch as the path is not worth playing anymore.

Another successful example of minimax with alpha-beta pruning algorithm is in checkers playing. Jonathan Schaeffer's Chinook program competed equally well against Marion Tinsley, 21-time world champion, in 1992 and 1994 Man vs Machine World Championship. Chinook employed a pre-programmed database for opening and endgame (with eight pieces or fewer) moves or, otherwise, a form of minimax search with alpha-beta pruning techniques. In 2007, Scheaffer et al. presented their proof that the best outcomes playing against Chinook is now only a draw [46].

2.2.1.3 Machine Learning Algorithms

Instead of a brute-force approach and pre-programmed database sequences, another approach is to learn game strategies from playing with humans or computers, including the machine itself. This is similar to a play-based learning practice in children education. Several algorithms from artificial intelligence (AI), computational intelligence (CI), machine learning (ML) as well as statistical learning are invented and implemented to find the appropriate solutions for each game challenge. For example, David Fogels used a neuroevolutionary algorithm to evolve the weights of Artificial Neural Networks with Evolutionary Algorithm in Blondie24, a checkers-playing program [33].

Recently, unlike Deep Blue's specific task machine with a brute-force approach, DeepMind's AlphaGo used deep neural networks, along with supervised learning and reinforcement learning, to learn from human-played and self-played games, whatever game it is [50]. This deep learning method astonishingly defeated Lee Sedol, the 2nd ranked international Go player, in March 2016. In 2017, DeepMind introduced a superior AlphaGo Zero and AlphaZero that can learn solely from self-playing by reinforcement learning, without human knowledge imposed beyond the game rules. Within 3 days of self-learning, AlphaGo Zero surpassed the ability of AlphaGo, that once beat Lee Sedol, by 100:0 wins. Generalized toward other games, AlphaZero played Chess and Shogi and Go then bettered the world champion programs in all three categories, after less than two days of self-learning.

Prior to this, the earlier versions of deep learning model from DeepMind company had already been capable of playing classic video games from Atari 2600, and had achieved the same level as a human professional game tester [36]. Currently, DeepMind is applying this method to play StarCraft II, a multiplayer, real-time, partially-observable strategy video game. The existing algorithms, however, played the game poorly, completely losing when playing against the hard-coded rule-based game bots provided by the StarCraft publisher.

2.2.1.4 Video Games and Machine Intelligence

Contrary to traditional board games, video games operate on a more complex and continuous space. With additional interactive elements, the dynamic and complication of game states hugely increase. In its early years, a video game often used a decision tree or a finite-state machine (FSM) to control its agent's state, action, or reaction [35]. A decision tree is a tree-like structure where a node acts as a decision point; an edge as a decision choice; and a leaf node as a decision. It is a very simple and fast, yet static, mechanism for decision making. We may include a random component into a node to break its predictable outcomes. On the other hand, FSM is a directed graph where a node represents an agent's state and a directed edge represents a transition from one state to another. An agent remains in its current state and carries out the same action, until a specific eventcondition occurs. Then, according to the event and its corresponding transition, the agent will change its state. We can combine other computation techniques with FSM to obtain some interesting features. For example, a fuzzy state machine applies fuzzy logic property into its components: a state transition triggered by a fuzzy logic, multiple current states with different degrees of membership, etc.

Unlike early-era video games, the complexity in contemporary video games limits the conventional use of a game tree and a state machine. Following the trend of modern-day video games, shifting toward the internet and mobile gaming, a real-time, massive-player game style is emerging. With very large branching factors and parallelly continuous game states, an adaptive learning approach has become much more promising [32]. This new method not only provides superior mechanism for game controllers, but also opens novel possibilities in game design, development, and analysis: exploring techniques to model human players' feelings or emotions; creating non-player character (NPC) controllers (that play much like real human players or play a game amusingly to attract audiences); building automatic systems to find exploits in games and help tuning the games toward specific intentions; finding procedural methods to generate game content suitable for a specific player, etc. [64]

2.2.2 Overview of Automatic Video Game Generation

Several attempts to automatically generate video games, both partially and entirely, have been proposed and implemented largely in the past ten years. With emerging machine intelligence algorithms and a strong demand in video game industry, the researches in this area continue growing rapidly. Among the earliest efforts to solve the problems, Nelson et al. viewed a game design as a problem-solving activity and classified it into four interacting aspects [37].

1. **Abstract game mechanics** specify abstract game states and game state transitions. The state transition dictates how a state changes from one to another, either by intrinsic conditions within a game or extrinsic interaction with a game player.

2. **Concrete game representation** specifies how to represent the abstract design in (1) to a game player in a game world. Generally, we map visual and audio elements to represent each abstract game state into the concrete game world. For example, we may represent a game time limit with an in-game watch, an on-screen bar graph, an action of a game agent, or an increasing tempo of background music.

3. **Thematic content** refers to the real-world references depicted on the game application. This is where the actual visual and audio elements are generated, as designed in (2).

4. **Control mapping** defines the relationship between player inputs and state transitions. There are several kinds of player inputs in modern video games: pressing a keyboard, pushing a joystick, saying a word, tapping a pad, moving a body part, etc.

In addition to the above design aspects, Liapis et al. viewed video games as another domain in computational creativity, among other existing domains such as music, story-telling, painting [30]. He also pointed out the unique characteristics of video games: highly interactive, dynamic, and content-intensive. It is a combination of these features that engages and entertains game players. Thus, the measures of feelings and emotions to quantitively evaluate game designs are required. Togelius et al. surveyed the theories of fun and curiosity, i.e. Csikszentmihalyi's concept of flow, Schmidhuber's theory of artificial curiosity, to measure fun in practice [57]. They applied these measures as a fitness function in a proof-of-concept experiment to evolve game rules for a single-player video game.

2.2.3 Automatic Generation of Game Rules & Mechanics

Game rules define a player's goals, including winning and losing conditions, and provide the player freedom to act or interact within a game. These two functions lead to the main structures for the game. With the help of game mechanics, action and interaction between a game and a player occur. Game mechanics also modify game data and transform game states. For example, in a platform game, the gravity is used as a key mechanic on a character control. A game player controls a character to run and jump over obstructive holes toward a destination. A game rule states that a character must avoid falling into a hole or the player loses the game. Automatic design of game rules and mechanics will help a game designer to create new games, or unique game genres. It may enhance a new idea, on a human design process, and exploration, for alternative solutions. The key concern for automatically generated games, however, is the playability: the player must be able to win a generated game.

Yavalath, a two-player strategy board game invented by Cameron Browne's Ludi system in November 2007, is the first computer-generated game to be commercially published [3]. Generated games in the Ludi system were described as symbolic expressions of game information units: number of players, board size, board shape, winning and losing conditions, etc. The system used genetic programming to evolve the sets of game rules in the self-play simulations. To search for new interesting games for human players, the concept of game quality is incorporated into an evaluation function. Major criteria in his game quality are related to playability tests and dramatic gameplays. The playability quality includes less draw outcomes, no unbalanced advantages toward the players, no too early serious disadvantages in the game, and suitable length of playing time. For the measurement of dramatic gameplays, the differences of estimated strength between players were recorded throughout the game. A close competition (less difference) or a comeback (winning after negative difference) is a key indicator for dramatic moments which the game designers are looking for.

An interesting remark on Ludi system is the creation of a game called Lammonthm, which is almost identical to Gonnect, the famous connection game [49]. The only difference is on a single rule that is encoded in one gene. That means Ludi system is only a mutation away from discovering a great game. However, the EC mutation process, which is influenced greatly by randomness mechanisms, is not guaranteed to perfect this incident.

Game rules are usually generated in a form of grammar-based compositions, constructing a complete game from elementary units according to grammatical constraints. In accordance to this approach, Video Game Description Language (VGDL) offers generic and flexible constructs to describe various kinds of video games. VGDL consists of four sections: level mapping, sprite setting, interaction setting, and termination setting. Video games defined in VGDL are used in General Video Game Playing Competition(GVG-AI), an annual competition of autonomous game agents, playing a group of both known and unknown video games for the machine intelligence algorithms. Having both automatic game-rule generations and efficient agent-controlled algorithms, this competition provides a solid ecosystem for advanced researches in video games.

2.2.4 Automatic Generation of Game Contents

2.2.4.1 Procedural Content Generation (PCG)

While well-designed rules and mechanics engage existing players for replays, carefully-crafted game content attracts new players by giving them irresistible influence of first impressions. Procedural Content Generation (PCG) is now a fast-growing area in both technical game research and commercial game industry. PCG is the algorithmic creation of game content with limited or indirect user assistance, either from the designers or the game players. In a broader sense, the term *content* refers to anything contained in games: characters, weapons, textures, music, maps, levels, stories, quests, game rules, etc. The only exception content in PCG research is the agent or game controller's behavior, which was discussed in the earlier section, due to its dominance in the research area. The unique characteristics of PCG, apart from other generative arts, are the playability and design constraints for a specific game genre.

PCG alleviates human designers' roles and accelerates game development, reducing costs while maintaining equivalent quality at lesser memory storage. It not only adaptively produces new content suitable for a player, but also inspires or collaborates with a designer in a mixed-initiative system. However, according to the current technology, game designers hardly use PCG in a real game production, due to difficulty in controlling the content creation, following a required direction within a short production time. There are severe tradeoffs between quality and speed as well as diversity and reliability [49].

2.2.4.2 PCG with Search-based Algorithms

PCG uses search-based algorithms, mainly evolutionary computation (EC), to search for good game content using a user-defined evaluation function. To create a successful search using EC, there are three major points to think about. (i) *The evaluation function* evaluates each candidate content then guides the evolution toward the optimal solution. The function should integrate intended features, e.g. playability, aesthetics, and intricacy into an account. Thus, defining a decent evaluation function is a challenging task. Different evaluation functions normally lead to different search results. (ii) *The content representation*, which is the landscape where an EC algorithm takes place, is also equally important. The content representation defines the search space for the algorithm and the way the searched content can be explored. (iii) *The search algorithm* itself has a crucial role in evolving results. Some algorithms are best suitable for a specific type of problems. For example, when there are multiple criteria opposing each other in decision making, a multi-objective evolutionary algorithm like NSGA-II is recommended [49].

2.2.4.3 PCG with Machine Learning Algorithms

Another approach in PCG, a worth mentioning one, is machine learning (ML) techniques, e.g. neural networks, Markov models, clustering, matrix factorization [54]. Trained by existing game content with supervised player response, PCGML is not only able to automatically or collaboratively generate game content variations based on training dataset, but also applicable to game content analysis as well as game parameter tuning. We also benefit from other features of ML including clustering, classification, identification, prediction, etc. Unlike evolutionary computation methods, where game content is converted into content representations, machine learning mechanisms directly use the content to learn from; no more conversion to an unfamiliar medium for a designer. This significantly lessens a game designer's work. Moreover, in order to use its ability to evaluate game content automatically, we can integrate PCGML with search-based PCG or other generative techniques.

Apart from the standard approaches mentioned above, some specific algorithms can also be applied to generate a particular type of game content [49]. In computer graphics area, fractal and gradient noise functions are used to create terrains, landscapes, textures, and cloud. Cellular automata are used in game-level generation, especially for dungeons and mazes [24]. Various kinds of plants can easily be created by Lindenmayer system (L-system). Planning algorithms are suitable for the creation of game quests and storylines [17]. Obviously, several existing techniques in other research areas are closely related to PCG in video games.

2.2.5 Automatic Game Parameter Tuning

2.2.5.1 Traditional Game Tuning with Human Playtesting

Game parameters are variables in a video game that directly control the players, enemies, and levels. They have strong influence on the game difficulty. Game tuning is a process to adjust game parameters, without modifying game rules and mechanics, to make the game fun, fair, and fascinating. In a video game production, this process is done under a testing stage, in which a game in development is played thoroughly to identify potential bugs and design flaws. Human testers provide feedbacks in a form of written surveys or verbal interviews; the data must be exhaustively compiled and carefully analyzed for necessary game tuning. Relying on iterative judgement, game designers use their experience and intuition as well as user feedback to adjust game parameters to reach the intended game difficulty. After all, a player's perception on game difficulty reflects hisher ability on playing that game, and vice versa. Used as a teaching tool, game difficulty is a key factor to train then evaluate a game player's specific skills.

Unlike dynamic difficulty adjustment (DDA), which automatically changes game parameters in real time based on the player's ability while playing, game tuning determines a set of values for game parameters to match a player's desirable level.

Generally, it is essential to set countable game content, e.g. the number of enemies, the maximum time to do a quest, etc. to constant initial values. (Increasing enemies' strength while fighting, due to DDA, is unnoticeable to a player. Increasing the number of enemies while fighting, however, is unfair.) Different game parameter settings bring different game variants, resulting to distinctive playing experience. Fine-tuning game parameters to achieve the game balance for generic game players, neither too easy nor too difficult, is laborious and time-consuming. Hence, we need automatic game tuning methods to alleviate this burden.

2.2.5.2 Automatic Game Tuning in Minimal Action Games

Isaksen et al. explored game parameter space on minimal action games [23], a game genre in which a player uses high skills on minimum control to play the game. He applied score probability distributions, in a form of survival analysis, on single-player *Flappy Bird* video game. By pressing a single button to emulate a bird's flapping, a player must navigate the bird through a series of pipes as far as possible. Each time the bird flies through a pipe gap without crashing, the player scores a point. Isaksen relied on these distance scores, as internal game matrices, to indicate the player's skill level. He created a player model based on human motor skills to imitate a human playing.

It is worth mentioning here that a player model is a general term 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 [51]. For automatic game parameter tuning, the player model is usually a static, objective, simulation-based, and player-experience model.

With a huge amount of time spent for automatic playtesting, Isaksen generated a histogram showing the number of surviving birds after flying pass each pipe. This presents a probability distribution of a player's scores for a specific game variant. By varying game parameters, he obtained numerous survival statistics from various game variants. With the survival analysis, he was able to understand the relationship between each game parameter and the perceived game difficulty. Using this technique, he explored game parameter space, looking for playable games, finding interestingly unique variations [22], searching game parameters for specific difficulty, etc. These applications are now really helpful for game designers to fine-tune game parameters effectively.

Isaksen's proposed survival analysis is useful for minimal action games, where its difficulty is determined by a player's motor skill. There are various kinds of game difficulty depending on a game genre. Picture puzzle games demand a player's visual skill and impose representational difficulty. Strategy games challenge a player's strategic skill, which can be measured by a deep look-ahead on a search tree, for example. Thus, as Isaksen concluded, various types of game difficulty require different models to accurately simulate and measure their effects [23].

2.2.5.3 Automatic Game Tuning in Two-player Action Games

The idea to use a player model for automatic playtesting is now proven as a solid approach for game tuning. Nevertheless, building a custom game agent for a player model is not an easy task and still time-consuming, not to mention a possible poor performance due to unforeseen game scenario. Liu et al. proposed to use now-available autonomous game agents designed for the General Video Game Playing competition (GVG-AI) in place of a customized controller [31]. This annual competition provides an ever-growing collection of autonomous agents for both single-player and two-player game tracks. The provided agents use several algorithms, ranging from a random number generator, genetic algorithm to Monte Carlo tree search (MCTS), as their controller. In her experiments, Liu used GVG-AI sample MCTS agents to play a two-player space-battled clone of *Spacewar* video game. The game is stochastic and fully observable.

Liu also suggested using the skill-depth of a game [27] as a fitness function in place of Isaksen's game difficulty. To demonstrate an automatic game tuning, she used simple evolutionary algorithms to optimize game parameters in search of game variants with high winning rates, which was used as estimating measures of deeper skill-depth. Alternatively, some other interesting fitness functions can be used in place of optimizing game parameters for skill-depth or game difficulty.

2.2.5.4 Automatic Game Tuning in Action-Adventure Games

Gaina et al. evolved game parameters for strategic diversity in an action-adventure clone of *Legend of Zelda* video game [19]. Games with high strategic diversity provide more paths to achieve the same goal than low strategically diversified games. Like Liu's experiments, Gaina used the same GVG-AI autonomous agents and evolutionary algorithm techniques. Interestingly, although this methodology produces positive results computationally, human subjective tests were unable to statistically differentiate such diversities. This may be the case that the GVG-AI autonomous agents played the game differently from strategically wise human players. Although computationally effective, the black-box model may contain low interpretability, making it incapable to understand the underlying mechanism.

For each research works discussed in subsections 2.2.5.2-2.2.5.4, we list some interesting features of the researches, including the target video game, the optimized game parameters, as well as key techniques for both player models and optimization process, in Table 7.1 in Chapter 7. The table also includes our methodology for game parameter tuning to be proposed in the next chapter.

2.3 Chapter Summary

In this chapter, we first describe computational intelligence (CI) techniques to be used in our framework and experiments, namely fuzzy logic system (FS), evolutionary computation (EC), differential evolution (DE), and coevolutionary algorithm (CEA). Afterward, we present the survey of related researches in the themes of artificial intelligence (AI) and CI applications in games, focusing on the automatic video game generation and parameter tuning.

Although automatic game parameter tuning is still an under-explored research field [31], there have been several decent papers continuously published during the past few years. It is interesting that those researches working on game parameter tuning, including our method to be proposed in Chapter 3, share one same idea in using simulated player models. With a suitable game-specific player model, we can automate game playing simulations and repeatedly adjust game parameters, using any appropriate AI or CI algorithm, to meet our intended objectives. The approach to study the relationship between game parameters and game difficulty truly helps game developers in their game tuning process. With in-depth knowledge in the correlation between game parameters and game difficulty, we can create more interesting video games and game contents which are not only entertaining for playing but also educative for teaching and learning purpose.

Chapter 3

Framework for Automatic Game Parameter Tuning

3.1 Introduction

In this chapter, we introduce our framework for automatic game parameter tuning. In a typical video game tuning process, we adjust some settings of a video game to make the game more fun for a wider range of game players. Our method focuses on the popular game genre called turn-based strategy (TBS).

The term *turn-based strategy* in TBS games describes a game containing two key features: turn-based and strategy. A *turn-based game* is a game where all players take turns when playing, while a *strategy game* is a game that focuses on a player's planning and decision makings to resolve the game situations.

It can be said that a TBS game depicts real-life situations where both determinations and fates play major roles to dictate an outcome. The structure of all TBS games is a combination of luck and strategy in various degrees. On the one hand, Tic-Tac-Toe, Scrabble, Chess, and Go depend entirely on a player's skills. On the other hand, Snakes and Ladders as well as most of other children games depend completely on luck. However, many of TBS games, e.g. Backgammon, Monopoly, Dungeons & Dragons, depend both on luck and skills. The familiar implementations of luck include rolling a dice, shuffling a deck of cards, and calling a pseudo-random number generator.

The outline of this chapter begins with the introduction of game parameter tuning process in section 3.2. Then, in section 3.3, we propose the framework of our automatic tuning process for TBS games. In the section, we explain the key components and their relationship in developing the framework. Regarding the player's side, we discuss a game player model, including how to evolve a player model, in section 3.4 and 3.5, respectively. As for the game's side, in section 3.6, we present how to adjust the game difficulty levels manually to create a player model that learns from playing easier games first. We also discuss a coevolution between a player model and game parameters in section 3.7. We summarize this chapter in section 3.8

3.2 Game Parameter Tuning

In this section, we first discuss about game difficulty and its relationship with a player's gaming skills in subsection 3.2.1. Based on the game difficulty, we then examine video game parameters in subsection 3.2.2. Subsection 3.2.3 explains the game parameters in TBS games, which are the target of our automatic game tuning framework. We finish this section with a conventional practice on a game tuning process in subsection 3.2.4.

3.2.1 Game Difficulty

Game difficulty is an experience that game developers design to challenge their target players in a game playing. Some games are intended to be more challenging for a player's learning experience, while others are expected to be less challenging for a player's relaxing experience. Game difficulty, depending greatly on a player's gaming skills, is a perceived experience that varies from players to players.

The theory of flow, illustrated in Fig. 3.1 (a), is a theoretical concept in Positive Psychology. Purposed by Mihaly Csikszentmihalyi in 1975, it explains mental states of a person with the focus on performing an activity. This theory is applied by Jenova Chen in his video game design to create the playing immersion or the flow state [6]. Figure 3.1 (b) illustrates the flow state in video game playing as the relationship between game difficulty and the player's ability.

According to Fig. 3.1 (b), a player has his or her own flow zone where the game challenge matches the player's ability. This zone is flexible and different for each player. A player maintains his or her focus in the game as long as the game balances its difficulty with the player's skills. On the one hand, once the player's ability increases beyond the zone, he feels dull and, finally, bored. On the other hand, once the game challenge increases beyond the flow zone, the player feels uncomfortable and, eventually, anxious. This concept, known as flow experience, is also used in Dynamic Difficulty Adjustment (DDA) system, where a video game adjusts its game difficulty adaptively, while being played, to maintain the player's engagement.

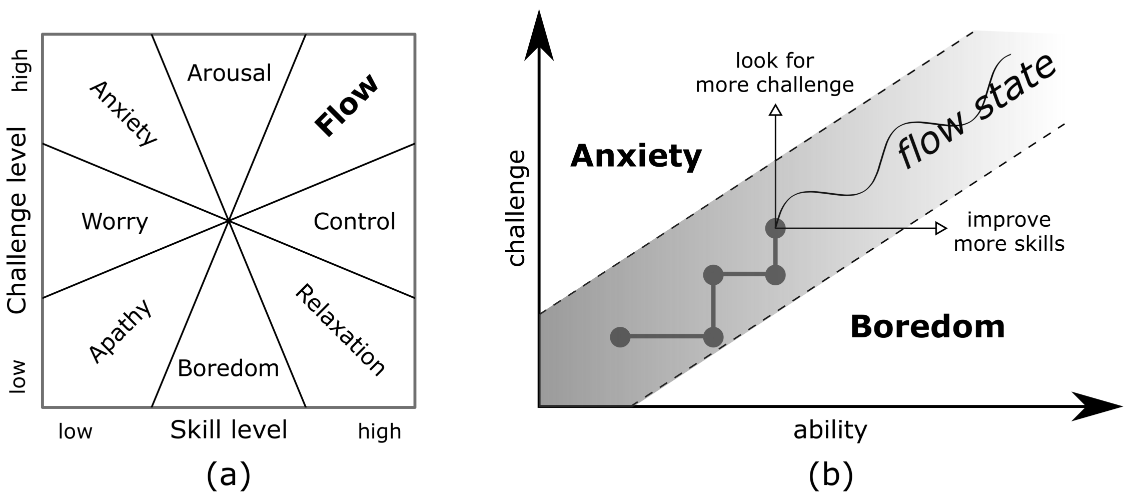


Figure 3.1: Theory of flow, (a) a concept about immersive mental states in Positive Psychology purposed by Mihaly Csikszentmihalyi, is applied to explain (b) the flow state in video game players. Jenova Chen's flOw video game uses this concept to engage players in the cognitive level.

3.2.2 Video Game Parameters

A video game contains a lot of parameters to adjust various properties of its looks and feels, e.g. player interfaces, game characters, background, sounds, motions, etc. We call a group of parameters that control game difficulty as *video game parameters.* Game parameters are internal constants or variables in the source code of a game program. They cannot be set directly by game players. It is a game developer's task to set these game parameters properly during the game development process to create a fun game.

Each game parameter contributes to game difficulty differently. Some game parameters, especially the ones directly related to the goal of the game, have obvious effects on the game difficulty; others may relate to game difficulty indirectly or partially to a certain degree. For example, for Star Trek game (see Appendix A), increasing the number of Starbases, which supply energy and weapons to the Enterprise, makes the game easier to play. However, increasing too many Starbases does not result in the game being much easier as the main goal is to destroy all Klingon's spaceships.

Game parameters are commonly related to one another. Changing one may affect the others. Moreover, we can achieve similar levels of game difficulty with many combinations of game parameter settings. This shared relationship of game parameters complicates the game tuning process. For example, for Star Trek game, either increasing the game time (Stardate) or decreasing the number of Klingon spaceships makes the game easier to win. The combination of game time increment and spaceship decrement both create easy games.

Basically, the game parameters being worth fine-tuning are the ones that closely relate to the main goals of the game. Therefore, we can adjust game difficulty more efficiently with these goal-oriented game parameters. They come in various forms, domains, and ranges depending on the game genres. Table 3.1 shows some examples of game parameters typically found in a TBS game. They are the subjects of our proposed tuning method.

3.2.3 Game Parameters in Turn-based Strategy Games

Although a huge number of TBS games are available in the market, they differ mainly in appearances, styles, and audio-visual designs. Most of them, however, share some common game rules and mechanics which are unique and specific to the TBS game genre. This helps reduce the learning curve when a gamer starts playing a new TBS game.

The main designs of TBS game genre focus on the planning and strategies of events or resources. To exhibit the events or resources in games, the representation of space and time are often created in the form of a map and a player's turn, respectively. The games initially set up the scenario, establish some conflicts, then assign the mission. It requires a player's skills in strategic thinking and decision making to accomplish the mission. The common TBS game design includes the following four elements.

- **Game Objects and Conflict**: Game objects are common elements in all TBS games. They are mainly classified into allies, opponents, and neutral game objects. The game normally sets up conflicts between the opponents and the game player, with assistance from the allies. Game objects contain their own actions and interactions which establish dynamics in the game.

  Typical game parameters for game objects are the quantity of game objects and the ability levels in each object, e.g. energy level, health score, weapon strength, etc.

- **Map and Exploration**: A game map displays the positional relationship among game objects as well as other important game states. In some games, a map terrain could be another type of game objects.

Similar to the board in a board game, the map in a video game is generally shown in the top view. This view provides a big picture and helps create more effective strategies. With advances in computer graphics, certain video games offer different views of a game map. For example, isometric, oblique and 3D views, which add depth information for a player.

For a game with a partially observable map, a player explores each area in the map to complete a given task. In such a game, map-related game parameters play a major role in adjusting game difficulty. The most common ones are map sizes, divisions, points of view, etc. Galaxy map in Star Trek game is an example of a partially observable, top-view map.

- **Resource Management**: Resource management or economic challenge is the unique characteristic of TBS games. Quantity and types of resources vary from game to game, e.g. energy, coins, vegetables, armies, etc. Some games emphasize on resource acquisition and conversion, while others focus on resource balance and usage.

  For resource management games, the resources themselves are the main game parameters. The frequently used parameters are the quantity of resources, conversion rate or exchange ratio, distribution of resources throughout the map, etc.

- **Constraints**: The constraints are the game mechanics to create excitement and control the challenges. The best-known constraint in games, including sports, is time. TBS games usually use time as (i) *a game time* to limit the number of player's turns or (ii) *an interval time* to limit the amount of time given in each turn. Usually, the amount of time is linearly proportional to the game difficulty, i.e. the longer period of game time, the easier to play and win the game, and vice versa.

We summarize examples of game parameters according to the above common design for TBS games in Table 3.1. We also show the corresponding Star Trek parameters in the last column. The game rules and mechanics of Star Trek game do not define conversion rate, exchange ratio, or interval time parameters in the game. For the current implementation of Star Trek game, we use a pseudo-random

Table 3.1: Examples of game parameters typically found in a TBS game, categorized by TBS game design elements. The last column shows the name of game parameters in Star Trek game. Two Star Trek parameters, marked by an asterisk, are selected to control game difficulty in our experiments.

Design Elements	**Game Parameters**	**Domain**	**Star Trek Parameters**	**Default Values**
Game Objects	quantity of game objects	discrete	*number of Klingons**, number of Starbases	*10 - 20* 2
	level of ability	continuous	initial energy	3000
Game Map	map size & map divisions	discrete	quadrants in a galaxy, sectors in a quadrant	8×8 8×8
Resource Management	distribution of resources	-	distribution of Klingons & Starbases	-
	conversion rate exchange ratio	continuous	-	-
	quantity of resources	discrete	torpedo	10
Game Constraints	game time	continuous	*mission time**	*30 - 50 (Stardates)*
	interval time	continuous	- (no limit)	-

number generator to distribute game objects throughout the galaxy map. Although we cannot control the distribution directly, we alter the random seed value of the generator to achieve a unique distribution of game objects.

3.2.4 Conventional Game Parameter Tuning Process

In a game parameter tuning process, game developers adjust game parameters to match their design objectives. The tuning process is a part of a *game testing* or *playtesting*, in which the video game is assessed systematically and improved extensively by a number of game testers and game developers to make it conform with overall requirements. In a conventional playtesting, human game players try playing the games in development then share their gaming experience and opinions. After that, game developers manually analyze the feedbacks and entirely improve the game product, not only the game difficulty but also the overall gaming experience. Due to the high cost of game testing process, game developers usually conduct this step close to the final stage of the production. This results in a very limited time to revise the game for major modifications.

3.3 Framework for Automatic Game Parameter Tuning

3.3.1 The Framework Components

The main task of game parameter tuning is to find suitable values for the key game parameters according to the playtesting assessment. We design our framework based on the conventional practice of the process. The proposed framework consists of two major components.

1. **Game Player Model**: A game player model represents a player in a video game. In our case, the player model plays the game in place of a human player. The model is game-specific and varies from game to game.

   The player model shares its underlying mechanism within the same game genre. By tweaking the parameters of this underlying mechanism, we obtain simulated player models with various levels of gaming skills. We use playing results from the skill-adjusting model and the video game as feedbacks to fine-tune game parameters.

2. **Mutual Evolution**: A mutual evolution is a joint development process between a player model and a video game environment, using an EC algorithm to balance between the player model's gaming skills and the video game's required level of game difficulty. We use the feedbacks from the interactions between the player model and the game to let them coevolve mutually until reaching the balance state in which there is no more improvement between the two.

   We can apply several techniques for the mutual evolution process.

   - Manual adjustment: The manual adjustment employs the investigative skills of game developers to adjust the game parameters. An example is a gradual incremental learning technique, explained in section 3.6. This method ensures that the end result of the modification serves their requirements well.
   - Automated adjustment: Guided by the interaction between the game and the player model, the method uses an algorithm to adjust the game parameters automatically. An example is a coevolutionary algorithm (CEA) technique, explained in section 3.7. This method eases the burden on

game developers and gives them more possibilities to discover extraordinary, either good or poor, results.

Therefore, each technique in a mutual evolution process differs largely on the contribution effort of the game developers and the controlling direction of the tuning results. In our proposed framework, we suggest a CEA technique as a mutual evolution mechanism for automatic game parameter tuning.

3.3.2 The Framework Structure

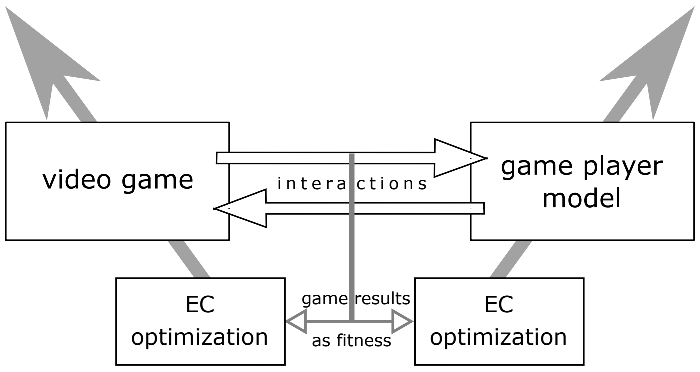


Figure 3.2: The proposed framework for automatic game parameter tuning process. The results from interactions between the video game and the game player models are evaluated as the fitness function for EC optimizations to coevolve both sides. The coevolutionary algorithm mutually evolves the difficulty-controlling game parameters and the multi-skilled player models. The game parameters are tuned to match gaming skills of a wider range of game players.

A diagram in Fig. 3.2 shows the overview of our framework. Starting in the top row, we let our player models play the video game repeatedly. Initially, the player models are weak and play the game poorly, when compared to the human players. We use the game results to improve the models iteratively by an EC algorithm. When the models improve, we have numerous player models with various gaming skills.

According to the results from playing with multi-skilled player models, the video game acquires perceived information of its game difficulty. The different game difficulty is the consequence of different game parameter settings. Therefore, we can examine the relationship between game parameters and the game difficulty within our framework.

In the mutual evolution process, the bottom row of Fig. 3.2, our framework adjusts game parameters to obtain the level of game difficulty that matches the simulated player models. Because video game interaction is a dynamic system, modifications in the game also lead to adaptations in the simulated players, and vice versa. The mutual evolution occurs repetitively between the difficulty-controlling game parameters and the multi-skilled player models. When the process reaches a balance, we are able to find the suitable values of game parameters that match the target level of game difficulty with various gaming skills automatically.

3.3.3 Comparison between the proposed Framework and Conventional Practice

The conventional practice employs a lot of professional playtesters to examine the game and generate intensive feedbacks through their experiences. Likewise, the automatic tuning process uses a large number of simulated game player models to play the game and generate the required data.

With simulated player models in place of human playtesters, we can separate the parameter tuning process from the game testing process. Instead of running the massive game testing process in the final stage of the production as shown in Fig.1.1, we can integrate the computational tuning process into a production at the earlier stage. We can perform the tuning process for any sets of game parameters independently, at any suitable time during the game production. The automatic tuning process lowers not only the testing cost but also the developing hours and other resources, e.g. recruiting human playtesters, documenting the feedbacks, etc. Furthermore, the process is capable to computationally examine shared relationship among the game parameters [23]; this type of analysis is difficult to obtain directly from human playtesters.

The main drawback of automatic game parameter tuning is that the technique underlying the effective tuning process highly depends on the game genres. Each game genre has its own characteristics and requires different gaming skills from a player. Some examples from related researches have been discussed earlier in section 2.2.5.

To investigate the feasibility and characteristics of our framework in details, we conduct several experiments to examine various aspects of the framework. The following sections present an overview of each experiment following the procedures of the proposed framework in our dissertation.

3.4 Player Model for Turn-based Strategy Games

A game player model is a computational model that represents 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 [63].

Because the framework focuses on TBS game parameter tuning, our player model represents human game players' behaviors in playing a TBS game. In strategy games, the main behaviors focus on a player's decisions. Therefore, we create the player model to replace a human player's decision-making process. Figure 3.3(a) shows a human game player playing a video game, while Fig. 3.3(b) illustrates a replacement of the human player by a game player model in our framework.

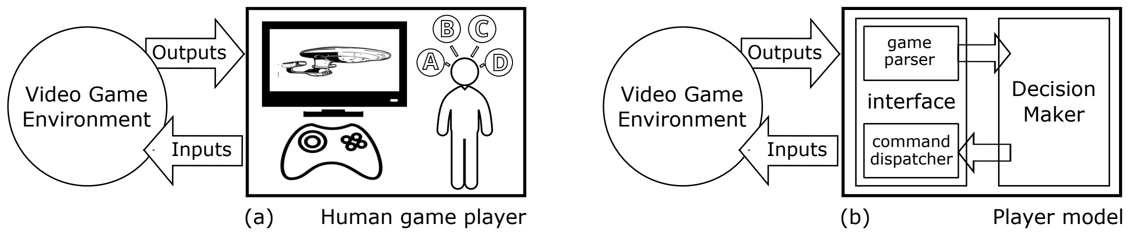


Figure 3.3: (a) A human game player is replaced by (b) a player model to create a simulated playtester for a game tuning process.

Our framework permits human game developers or skilled players to integrate their game expertise in the tuning process. They use their expertise in TBS games to design more efficient decision-making process which can play the game better. Our framework supports their knowledge transfer by providing decision-making tools that is convenient for them to use.

Our game player model uses a computational intelligence approach for the knowledge-based decision making. Fuzzy logic system (FS) allows an expert player to express his or her game decisions in human terms using Fuzzy rule-based system (see subsection 2.1.2.5). This FS approach establishes an FS decision-making system for our TBS game player model.

In addition, the approach allows the game developers to tweak its parameters to produce numerous player models of various game-playing performances. This results in the simulated players of various gaming skills.

The FS decision-making system may not provide the best decision when compared to other more sophisticated techniques, e.g. neural networks or deep learning approach. However, its decisions are practical enough to use when playing against typical TBS games with fair successes. Nevertheless, our main objective is not to find the best player to play against computers or humans. It is a variation of gaming skills of the player model, representing a wider range of game players from novices to experts, which is more important in our framework.

The benefit of the FS rule-based system lies in the fact that the given FS rules are human-understandable. Unlike other techniques of the black-box model, in which the knowledge of its internal mechanisms is unknown, the FS rule-based approach is easy for an expert to create, simple for a developer to maintain, and straightforward for a person to understand. This is the key point in using the FS rule-based system in our methodology.

We present additional details on the game player model in Chapter 4. The chapter explains the motivating concepts behind the design of a player model. It also demonstrates a sample implementation of a player model for the TBS game named Star Trek.

3.5 Evolving Fuzzy Logic Rule-based Player Model

In a conventional method to create an FS rule-based system, a domain expert provides both FS rules and FS membership functions for a fuzzy knowledge base. FS rules relate the game's fuzzy conditions to the expert's decision while FS membership functions control the interpretation of fuzzy conditions. Adjusting membership function parameters to control the function shape for the best decision is a tedious and laborious task. Therefore, we use an EC algorithm as an optimization tool to search for the optimal values of membership function parameters.

It has been confirmed that the player models with EC-optimized FS membership functions perform better than some manually adjusted ones. We present our experimental results to support this claim in Chapter 5. The chapter also discusses more details on the evolving player model.

An EC algorithm in our framework lessens an expert's tasks on specifying FS membership functions. It pushes the system one step ahead toward an automatic framework. The expert, however, still has to specify the FS rules to guide the system in the preferred direction. The given rules also exhibit the expert's behavior characteristics, e.g. aggressive, preemptive, cautious, etc., which are various among the experts.

Although the automated generation of both FS rules and membership functions is possible, the search space would be enormous. Furthermore, it is not easy to control the number of FS rules or input variables in the automated rule generation [28]. In addition, automatically reproducing a specific behavior is also complicated. Having both expert-generated FS rules and EC-optimized FS membership functions is a more practical and manageable procedure for our proposed framework.

3.6 Learning Player Model by Gradually Increasing Game Difficulty Levels

Once a player model is strong enough, equipped with a variety of gaming skills, our framework is ready to perform the mutual evolution between the player model and the video game environment. To be precise, we are interested in the balances between gaming skills of the player model and game difficulty of the video game environment, i.e. the player model parameters and the video game parameters.

We propose a technique called *gradual incremental learning* to manually adjust game parameters for a mutual evolution process. We have an assumption that, to train a player model for a specific game difficulty, it should be more efficient to start training at the lower difficulty level. Once the model evolves and gets more skilled in these easier games, we then advance the game difficulty. We keep adjusting game parameters gradually and continue training the model continually toward the specific level. We believe that this approach can improve the model better than training the model at the target level only. This idea is similar to the incremental learning concept in machine learning, where a learning model adapts to new data without forgetting the existing knowledge.

Our experiments in Chapter 6 show how the player model successfully adapts to the gradually increasing game difficulty. Its performance at the target level of game difficulty has obviously improved. This is considered a half-way success toward the mutual improvements with a CEA technique. That means our idea for automatic game parameter tuning with CEA techniques is promising. We discuss more details on the gradually incremental learning of game difficulty along with the experimental results in Chapter 6.

3.7 Coevolving Game Parameters with Player Model

Unlike any other EC methods, CEA requires no fitness evaluation on its population. Instead, we have to provide the controlling policies for the algorithm. Basically, the policies focus more on the interaction among groups of population. It depends largely on the feedback from one group to another group. With these policies, we can roughly guide the direction of mutual evolution to meet our requirements.

There are two policies of major concern in CEA approaches:

- **The relative fitness evaluating policy**: This policy specifies how we evaluate the overall game-playing results. In our case, we require two complementary policies: one policy for the player model when playing various-difficulty games and another policy for the game when being played by multi-skilled player models.

  The design of the policy depends largely on the objectives of the tuning purpose. For example, a game for training purpose may require a higher level of difficulty than a casual game. In this case, we may add some biases toward winning more games in the policy.

- **The sampling policy**: This policy specifies how we sample some individuals from a pool of population to interact with individuals from another pool. We control the diversity of the game player's abilities with this policy. For example, we can put biases toward lower-skilled player model when tuning children games.

3.8 Chapter Summary

We propose a framework for automatic game parameter tuning using a game player model. Our framework focuses on tuning a turn-based strategy game for players with diverse skills. We create a game player model as a simulated game player to play a TBS game. The core of the player model is the FS rule-based decision system, where a game developer or a skilled game player incorporates his or her expertise. In addition to an expert's knowledge, we use an EC algorithm to optimize the player model for a better performance. When the EC-optimized player model is strong enough, we let the model coevolve with the game mutually using a CEA algorithm. This is an iterative interaction process where a feedback from the game helps improve the model, and vice versa. When CEA reaches its balance where there is no more mutual improvement, the player models with wide ranges of game ability can play some games equally well. It is the key parameters of these games that produce the game difficulty which is suitable for human players with various gaming skills.

Chapter 4
Game Player Model

4.1 Introduction

In this chapter, we thoroughly discuss our player model in terms of concepts and implementation. To be more specific, we focus on how our player model represents a player in a turn-based strategy (TBS) game.

The role of player models, fundamentally, is to manipulate a game or keep the game designers informed about a game player's experiences or behaviors [29]. However, modeling a game player to assist video game designers, also known as modeling designer, is still in its early stage [63]. This type of player models can be viewed as a second-order player model [29]. That means a player's behaviors, simulated by the player model, are evaluated in order to aid the game designers.

Game player model is an umbrella term to represent specific information obtained when a game player interacts with a video game. There are many kinds of player models with different purposes, scopes of application, sources of derivation, and domains of finding [51]. Our model can be described as an *individual analytic action generator* following the terminology of Smith et al. [51]. This is a model that generates game inputs for an individual player, using an automated method that examines the mechanisms of a game. In our case, a fuzzy logic system (FS) is used as the analytical tool to generate the player's actions.

After this introductory section, we first examine interactions between a video game environment and a human player in section 4.2. After that, we use what we learn to model a simulated game player. We detail our idea on a game player model in section 4.3. In section 4.4, we suggest several methods to improve the player model's performance to best catch up with the levels of game difficulty. With such improvement, we can also generate player models with different skills needed when playing a game. To make a concrete example, we show our implementation of how a player model play a spaceship fighting TBS game called Star Trek in section 4.5. Lastly, we finish the chapter with the summary in section 4.6.

4.2 Interaction with Video Game Environment

In this section, we consider, in abstraction, the interactions between a video game environment and a game player in subsection 4.2.1. We conceptually examine how a player makes decisions in games and how game responses in subsection 4.2.2 and subsection 4.2.3, respectively. The relationship between the decisions and responses establishes a foundation of our player model framework.

4.2.1 Game States and Interaction Flows

We design our player model based on the interaction model between a video game player and a video game environment. We consider the video game environment as a set of game states being processed interactively by a three-action loop. The three fundamental actions in the loop are:

1. **Input:** A video game receives commands from a game player to manipulate game states via game inputs. A game input consists of a *command* and optional *command arguments*. The command argument identifies how to perform or the degree to operate the given command.

2. **Update:** The video game then updates its states according to either the external inputs from the player or its internal mechanisms from the game rules. The modification of a game state can be done through a decision-making system in the game, or a pseudo-random number generator, or the mixture of both.

   We use a pseudo-random number generator to construct a set of random numbers that is controllable to a certain degree via a random seed. These pseudo-random numbers retain the indeterministic behaviors when replaying the games.

   Making decisions in games is a large subject of both conventional practices and on-going researches. Besides the random decisions generated from pseudo-random numbers, an overview of other decision-making techniques in video games are briefly presented in subsection 2.2.1.

3. **Output:** A video game responds to its game player with updated game states through game outputs. For a typical video game, the term *display* can be used interchangeably due to the visual nature of the media.

As for the game player's side, the same procedures are followed in the interactive manner. That means the player receives responses through game outputs, makes or updates a decision, converts the decision into game command(s), then send the command(s) back as game input.

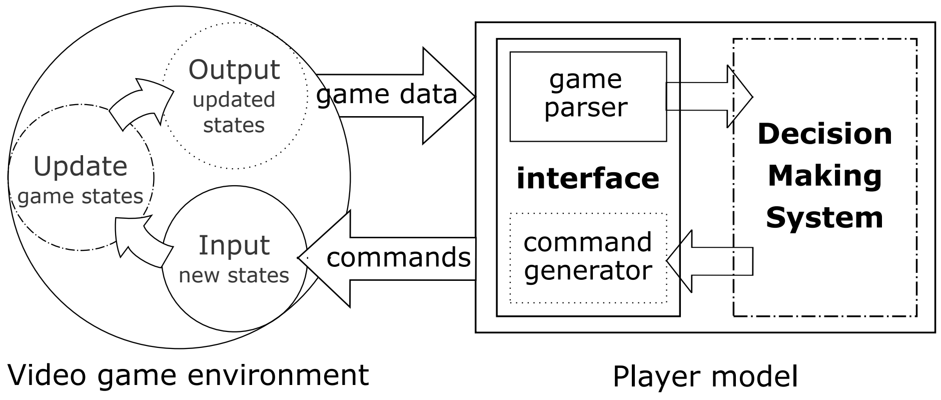


Figure 4.1: A model of interactions between a game environment and a game player.

The entire loop of the interactions between a video game player and a video game environment is presented in Fig. 4.1. This loop also represents the flow of game states that connects the two sides of decision makings. The loop iteratively and interactively continues until the game ends.

4.2.2 Hierarchy of Game Decisions

In a video game playing, we model a player's decisions in a top-down approach, from a conceptual idea down to concrete game commands. This hierarchy of the player's decisions consists of:

1. **Decision:** According to current game states, a target decision is initiated from a strategic plan to achieve the game objectives.

2. **Action Plan:** A series of actions to accomplish the target decision is then developed. For other games, besides turn-based games, the plan also includes a timing to execute each action sequentially.

3. **Game Commands and Arguments:** Game commands, along with their corresponding command arguments, are constructed to control the game states according to each action plan.

Once game commands are realized, a player inputs them to the game environment at a particular time. (This is also where the challenges of action games reside.) This triggers a new loop of interaction flows as described in previous subsection.

4.2.3 Levels of Game Information

After updating game states according to the player's inputs and the game mechanics, a video game sends out game data that can be classified into *game states* and *game information*. Game information is the direct response from the game sent to its player, in correspondence to the player's decision hierarchy described in the previous subsection.

We classified game information with a bottom-up approach, from command feedback up to decision calls. The levels of game information, therefore, consist of:

1. **Command Feedback** is a quick validation made to game command inputs or their arguments. It includes, but not limited to, a notice informing a newly given command is invalid, a hint showing additional arguments are required, or a signal warning the time is almost over for the next command. It is a low-level, command-related game information.

2. **Action Result** is when the game states are updated in response to an issued action command. This indicates a progress towards or a recess away from a player's target decision. It is a middle-level, action-related game information. The player should assess the action result and determine the next command to issue appropriately. Depending on this information, the subsequent commands are either a follow-up in the action plan or a new action due to unexpected changes of game states.

3. **Decision Call** is a request for a new game decision which can be either a direct or an indirect call. It is a top-level, decision-oriented information. One example of an indirect call is a critical warning [59]. This call requires an immediate resolution from the player. Without a suitable reaction, the player's game states may become worse, and the player may even lose the

game. The critical warning indirectly forces the player to re-evaluate his/her current action and decision.

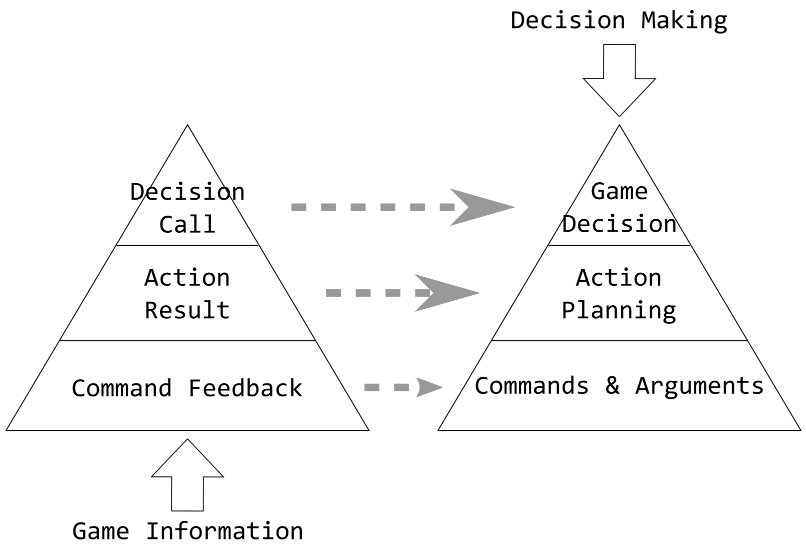


Figure 4.2: Association between hierarchy of game decision and levels of game information.

We illustrate the relationship between the hierarchical levels of game decisions and their corresponding levels of game information in Fig. 4.2. We use this relationship as the underlying mechanism for our game player model, which is explained in the following section.

4.3 Structure of Game Player Model

We use a game player model to simulate a human game player. Each component of the model imitates the way a human player perceives and reacts, both consciously and subconsciously, to the game environment. The main components and subcomponents of our player model consist of:

1. **The Game Interface:** Our player model communicates with a game environment through the game interface, as previously shown in Fig. 4.1 (b). For the input interface, it has *game data parser* to manage game input before making game decisions. For the output interface, it also has *Game Command Generator* to handle the game output before releasing to the game environment. For further explanation, we describe data parser and command generator modules in subsection 4.3.1 and subsection 4.3.3, respectively.

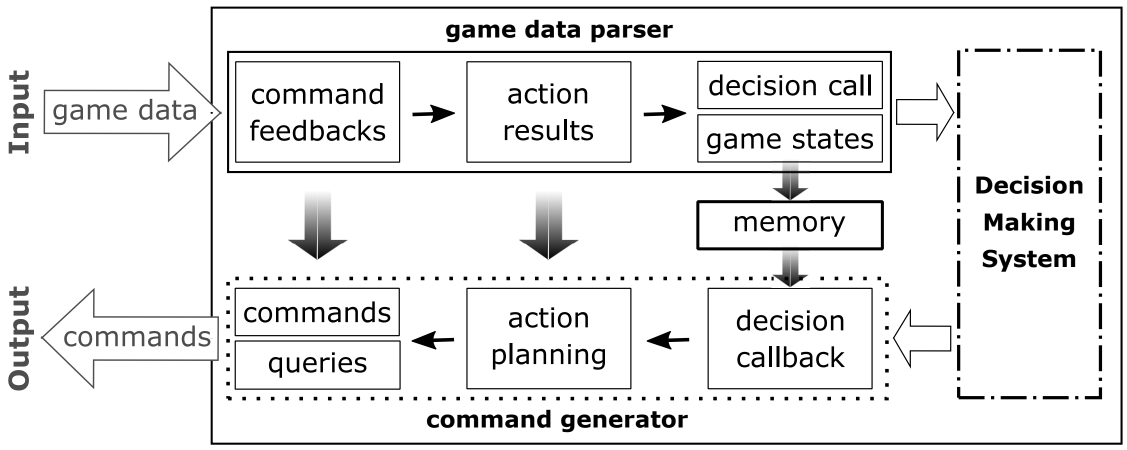


Figure 4.3: Game interface consists of data parser and command generator modules.

Even though both modules communicate with different parts of the game environment, they work closely with each other to, as efficiently as possible, handle the responses sent back to the game. This also helps avoid unnecessary calls for the computation of decision making. Fig. 4.3 shows the internal elements of each module and the relationship between them.

2. **The Decision-Making System:** The significant factor to determine the efficiency of our player model, especially the one for TBS games, lies in the design of *decision-making* module. The system makes a game decision according to game states given by the data parser then outputs the decision via the command generator. We explain the detailed mechanisms of the game decision-making module in subsection 4.3.2.

As illustrated in Fig. 4.3, the data flow in our player model starts with the input data from a game environment. Analyzed by the data parser, the game state data is supplied to the decision-making module for a game decision. Being a response from our model, the decision data is given to the command generator then converted into the game command. Finally, the command data is produced then forwarded back to the game environment. We discuss three modules of a player model in the following subsections, following the order of operation flow from inputting game data to outputting game commands.

4.3.1 Game Data Parser

Initially, when a video game delivers its game output, we need a way to extract necessary game states and information from the output into a new form that is suitable for our game player model. We obtain a game output then turn it to be an input for our player model. The terms *game output* and *player model input* are used interchangeably in this dissertation.

An output format of a video game varies in many forms. Due to the limited technology in the early stage of video game development, most of the games are text-based format, either in a printed document or on a cathode ray tube (CRT) monitor. The visual video game outputs became more popular when the display device technology advanced in the later era of video game development. The gaming displays improve from low-resolution screens in the 1980s to high-resolution LCD monitors in the mid 2000s and virtual reality (VR) headsets in the mid 2010s. Along with the better display devices, also came the better audio systems. The game sound is simply no more an auxiliary part of the game graphics. A lot of crucial video game information is conveyed in the auditory form. Another type of practicable game outputs is haptic communication. Haptic game devices recreate the sense of touch by applying force, vibration, or motion to a player. There are many opening opportunities for new game outputs with the advent of new technologies.

Because each video game environment uses various output formats, the ways of information extraction vary accordingly. Hence, the name of this module also differs due to the techniques used in the extraction method. As we aim to create a player model for text-based strategy (TBS) games, we use a parsing programming function to isolate between words looking for the expected information. We call our information extraction system a parser. Therefore, we use the term *game data parser* to refer to the text-based extraction method for our player model.

We wish to make an accurate player model in which its performance depends solely on its ability to analyze an actual data and reason the proper game decisions, without any support or bias from the game environment. Our data extraction works on the identical data given to a human player. This ensures that all decisions are made on the same raw data. Using different data sets is unfair and must be prohibited.

In the case of TBS games, our game data parser first separates the game states and information from incoming texts. After that, we assess the results of the previously issued game commands from the parsing information. When the latest commands succeed and a new decision is required, we supply the game states to the decision-making module for the next decision.

To assess the past commands from the game information, we examine the parsing information in order from the low-level command feedbacks up to the request of a new decision, as described earlier in subsection 4.2.3. The main purpose of the assessment is to quickly response to a bad decision as soon as it is noticed at the right place and the precise circumstances.

When a failure occurs at the command level, we may issue a wrong command or wrong command arguments. We can examine the command feedback information to change the command or to recalculate the command arguments, without modifying the existing action plan and the target decision. Similarly, when a failure occurs at the action level, we can examine the action result to abort unnecessary actions or rearrange the order of the action plan, without modifying the existing target decision.

The data parser module separates the main logic behind a decision making from the game output format. When the game updates its output format, which is very likely for a game in development, minimal changes can be applied to reuse the existing player model. Likewise, the improvement of a decision-making module has no effects upon the data parser module. This is the benefit of a modular design in our system.

4.3.2 Game Decision Making

The brain of our player model is the decision-making module. For a strategy game, it analyzes the current situation in the game (possibly with the help of memory to recall the pass events), synthesizes all feasible efforts, and build the best strategy to win the game.

There are various techniques in decision making. The techniques in game decision vary from simple methods, e.g. a decision tree or state machine, to complicated ones, e.g. Markov system or rule-based system [35]. They are applicable to both intra-character and inter-character decisions.

For a player model, it is typical to choose a decision-making method of which its complexity matches the complication of the game characters' behaviors. A state machine suits a game character with few strict behaviors. The rule-based system, on the other hand, matches a larger number of flexible behaviors.

As for our framework, we select a fuzzy logic rule-based system as our decision making. The fuzzy logic rule-based system for decision (FS decision making) uses fuzzy logic in describing the controlling rules to make decisions.

In this approach, an expert player creates an FS decision-making system by expressing their decision-making knowledge linguistically with fuzzy logic rules (FS rules). The technique requires less effort for a non-technical professional to present his/her expertise in the domain. It fits our main objective to use the insights of the game developer to create the player model. In addition, thanks to its human-like terms used to describe the logic of the rules, FS rules are easy to create, understand, and maintain. This characteristic is appealingly used in a production period under a team development, where changes from team members are common.

A player model with FS decision-making system may not establish the strongest player model for TBS games, when compared with other sophisticated algorithms, e.g. neural networks or Monte Carlo search tree. Our goal to create a player model is to produce a simulated game player which is suitable for game parameter tuning. The model is not, however, expected to be too powerful to outdo human players' performance. To meet this purpose, the FS decision-making system is efficient enough to produce just fine decisions for TBS games. With an optimization process introduced in Chapter 5, we can make the stronger player model that is capable to play the games more proficiently. In addition, due to its high level of interpretability, FS rules could provide the game developers opportunities to discover new findings from the optimized player model as extra advantages.

Based on the organization of a fuzzy rule-based system in Fig. 2.2 from subsection 2.1.2.5, our FS decision-making system consists of three major components as illustrated in Fig. 4.4:

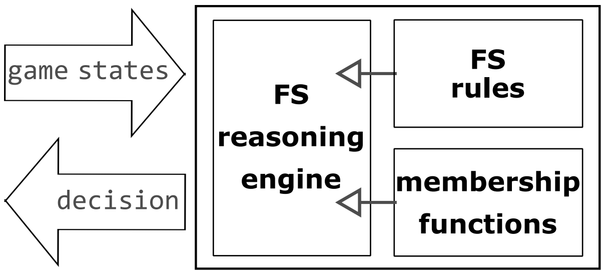


Figure 4.4: The decision-making module in our player model.

4.3.2.1 Fuzzy Logic System Rule (FS Rule)

FS rules specify relationship between game states, in the form of FS input variables and Boolean input variables, as well as their consequent game decision(s). A game developer or an expert game player creates FS rules from his or her expertise in game playing. The rules are their logical knowledge for the playing character's decisions.

It is easy to view the rules in a form of a table. We combine several rules sharing the same objective into a table, called an FS table, of which the size exponentially grows according to the increment of additional input variables. Appendix B and C show FS tables used in our player model for Star Trek game. Notice the relationship between the input variables and the size of the table in those sections.

4.3.2.2 Fuzzy Logic System's Membership Function (FS Membership Function)

FS membership function is an interpretation of an FS input variable used in an FS rule. The interpretation is dictated by the shape of the function which we control with function parameters. By altering the interpretation of FS inputs via these parameters, the output decisions change accordingly.

In a conventional practice, a game developer or an expert game player manually adjusts membership function parameters to tweak the given FS rules for the required outputs.

Alternatively, this FS membership function parameter is the subject of an EC optimization process to automatically improve the FS decision-making system. We present the technique in Chapter 5 along with the experiments to confirm the statement.

4.3.2.3 Fuzzy Logic System's Reasoning Engine (FS Reasoning Engine)

FS reasoning engine examines current values of game states according to the given FS rules and membership functions, then calculates the player's decisions accordingly. The reasoning engine performs the fuzzification, inference, and defuzzification processes of the fuzzy rule-based system to obtain the output decisions.

The game data parser module supplies the parsed game states to FS reasoning engine. With the given game states, the engine interprets the corresponding FS input variables into a degree of membership (DOM), following the matching membership function. According to the operators working on these input variables, a weight for each rule is calculated and assigned to its output. Then each output accumulates its weight from the rule to which it belongs to in the table. Eventually, the decision with the maximum weight is the output decision for the table. This decision is sent to the next module, the game command generator.

4.3.3 Game Command Generator

The main task of the game command generator is creating the output interface to the game environment. The game environment obtains the output from our player model as its input. Therefore, the terms *player model output* and *game input* are interchangeable.

Similar to the game data parser module, the modular design of the game command generator allows the separation of the command output from the main logic in the decision-making module. When the output formats are revised during the game production stage, we simply update the output interface in the command generator module appropriately, without touching the logics in decision making.

The game input format and its interface vary from one game to another. There are more varieties of game input devices than output devices. Notable forms of game input include a punch card of a mainframe computer in the 1970s as the primitive game input, a keyboard as a universal gaming device that allows many operations from a single press to a text typing, a mouse for point and click actions, a gamepad or a joystick for precise control and ergonomic handling, a touchscreen for convenient usage, as well as special-purpose devices for a specific game genre, e.g. a steering wheel and a pedal for racing games, a light gun for first-person shooting games, a guitar and drums for rhythm games, a foot pad for dancing game, etc.

As for TBS games, a game player inputs a text command to the game via standard input (stdin) with a keyboard. However, the communication between our player model and the game environment uses a text buffer that is directly connected. To send a command to the game environment, our player model directly prints a text command to the text buffer of the game.

Based on the hierarchy of game decision explained in subsection 4.2.2, the game command generator successively creates two keys hierarchical components: *an action plan* and *game commands and arguments.* With the resulting decision from the decision-making module, the command generator analyzes the given decisions and creates a series of actions, or an action plan, to achieve the target decision. The generator then maps each action in the action plan to the available game commands and calculates the command arguments accordingly.

We implement the game command generator, including the sequence of action planning and command mapping, as a decision callback function. Each output decision from FS decision tables is a reference to its matching callback function.

The callback function creates a list of actions for a specific goal. The function then maps each action into a game command available in the game environment. It also calculates a command argument when necessary. The callback function returns an array of game commands along with the objectives. We later use these objectives to verify its success with the command feedback and action result in the game data parser module.

In general, there are two types of game commands:

- **An action command** that modifies a game state.
- **A query command** that retrieves a current game state.

To make a good game decision, we must gather necessary game states thoroughly and accurately. We collect the required game states either by querying directly from the game environment or deducing them implicitly from other game states.

We also provide a memory to keep track of queried game states. With this tracking memory, our model can access a history of game information for better decision analysis and command argument calculation. We store essential information, including game states as well as the issued commands and their corresponding game responses for each move, to a memory internal to the player model. Accessing to the past information from this memory is an advantage for a player model over a human player's restricted memory.

One of the challenging tasks in a decision callback function is to provide a value for a command argument. There are many ways to create the command argument, e.g. allocating to a constant, assigning to a random value, searching with an optimization method, computing from fuzzy logic system, calculating from current and past game states, etc.

In a TBS game, when a game environment asks for a player's turn, the game command generator sends out the generated commands in its array sequentially. In other kinds of strategy games, the generated action plan must include a scheduler to issue the right command at the right moment.

4.4 Improvements to Game Player Model

Even though a game player model is easy to create in a paper, designing a working model for TBS games, especially for the complex ones, is quite a demanding effort for a novice. It is also a delicate work for an expert to create an optimal one without any helping tools. Therefore, we can assist the model creation with some tools.

The main target to facilitate the design and improve the performance of a player model for TBS games is bettering its decision-making module, where all game decisions and game commands are generated. With our use of FS decision making system, we introduce additional features to enhance the design of FS rules in the following subsections.

4.4.1 Modular FS Tables

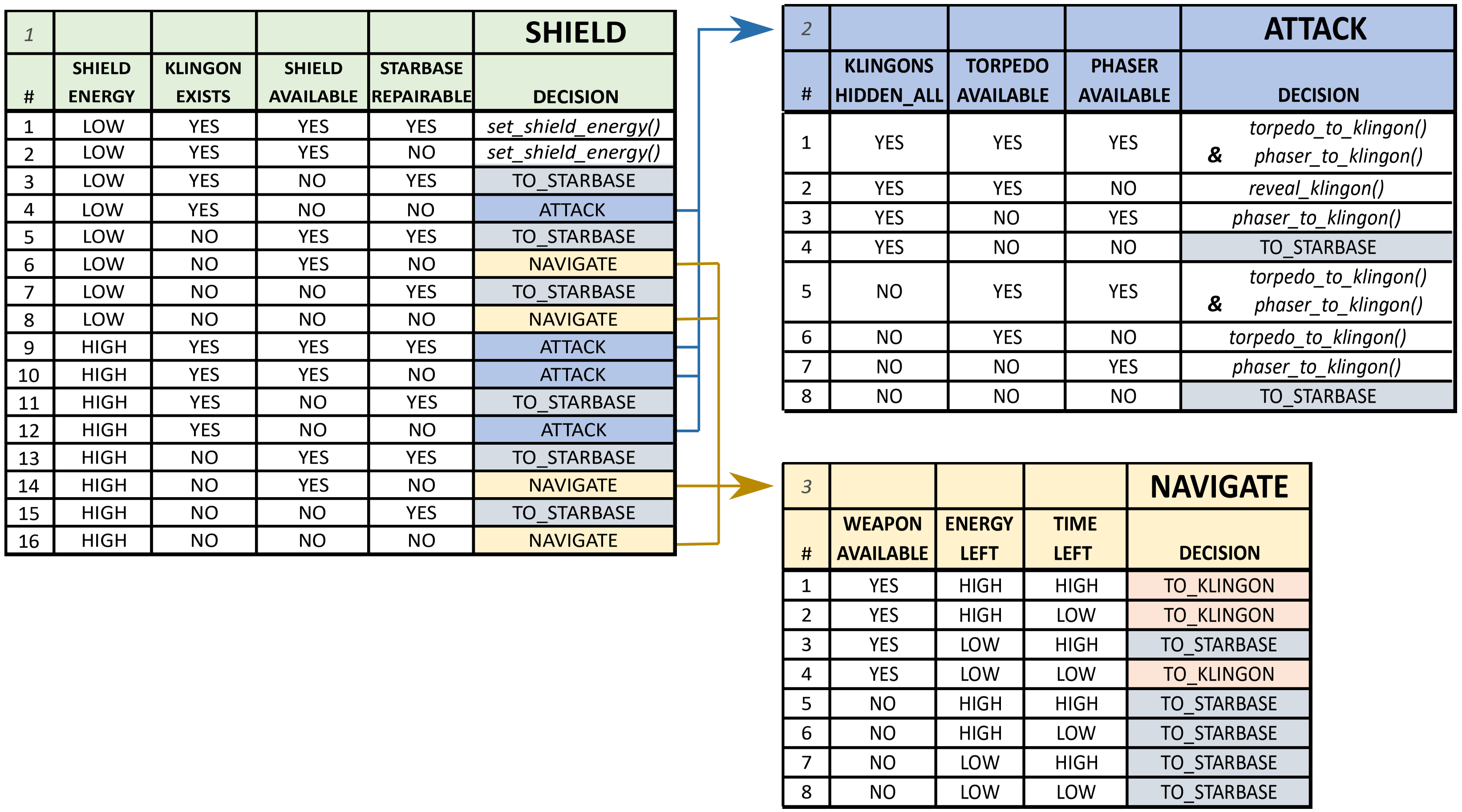
One of the well-known drawbacks of an FS rule-based system is the exponential growth of its table size. The size of an FS table equals to the multiplication between a number of degree of membership (DOM) states from each input variables. For example, an FS table with three inputs contains eight rules at a minimum, with two DOM states in each input variable (i.e. LOW & HIGH). However, with five DOM states in each input (i.e. VERY_LOW, LOW, MEDIUM, HIGH & VERY_HIGH), the table size rises up to 125 rules ($= 5 \times 5 \times 5$) rapidly.

To determine a fine decision for a strategy game, we need to consider a substantial number of input conditions, each with numerous DOM states. The size explosion problem is unavoidable when constructing such an FS decision-making system. One gigantic FS table is not easy to create and quite tricky to maintain.

To solve the problem, we iteratively split the game decisions into several smaller missions. Each mission concentrates on a single job and retains its own FS table, which we call an FS modular table. The output of the table is either a game decision or a pointer to another table. The whole connected tables, from one to others, establish a hierarchical structure of modular tables. A series of fuzzy reasoning must be computed before reaching the final decision.

The modular FS table drastically reduces the table size by dividing a huge FS table into small connected modular tables. A lot of FS tables, each with a few inputs, are easier to create and maintain than an extremely large one with considerable inputs. We can modify an FS modular table of a single task with a minimum effect to the others. Similarly, the extension of the existing modular table is convenient. The modular design also makes the underlying logics behind the overall decisions more understandable.

Figure 4.5 shows an example of our implementation of FS modular tables in



1					SHIELD
#	SHIELD ENERGY	KLINGON EXISTS	SHIELD AVAILABLE	STARBASE REPAIRABLE	DECISION
1	LOW	YES	YES	YES	*set_shield_energy()*
2	LOW	YES	YES	NO	*set_shield_energy()*
3	LOW	YES	NO	YES	TO_STARBASE
4	LOW	YES	NO	NO	ATTACK
5	LOW	NO	YES	YES	TO_STARBASE
6	LOW	NO	YES	NO	NAVIGATE
7	LOW	NO	NO	YES	TO_STARBASE
8	LOW	NO	NO	NO	NAVIGATE
9	HIGH	YES	YES	YES	ATTACK
10	HIGH	YES	YES	NO	ATTACK
11	HIGH	YES	NO	YES	TO_STARBASE
12	HIGH	YES	NO	NO	ATTACK
13	HIGH	NO	YES	YES	TO_STARBASE
14	HIGH	NO	YES	NO	NAVIGATE
15	HIGH	NO	NO	YES	TO_STARBASE
16	HIGH	NO	NO	NO	NAVIGATE

2				ATTACK
#	KLINGONS HIDDEN_ALL	TORPEDO AVAILABLE	PHASER AVAILABLE	DECISION
1	YES	YES	YES	*torpedo_to_klingon()* & *phaser_to_klingon()*
2	YES	YES	NO	*reveal_klingon()*
3	YES	NO	YES	*phaser_to_klingon()*
4	YES	NO	NO	TO_STARBASE
5	NO	YES	YES	*torpedo_to_klingon()* & *phaser_to_klingon()*
6	NO	YES	NO	*torpedo_to_klingon()*
7	NO	NO	YES	*phaser_to_klingon()*
8	NO	NO	NO	TO_STARBASE

3				NAVIGATE
#	WEAPON AVAILABLE	ENERGY LEFT	TIME LEFT	DECISION
1	YES	HIGH	HIGH	TO_KLINGON
2	YES	HIGH	LOW	TO_KLINGON
3	YES	LOW	HIGH	TO_STARBASE
4	YES	LOW	LOW	TO_KLINGON
5	NO	HIGH	HIGH	TO_STARBASE
6	NO	HIGH	LOW	TO_STARBASE
7	NO	LOW	HIGH	TO_STARBASE
8	NO	LOW	LOW	TO_STARBASE

Figure 4.5: Three out of five modular FS tables used in Star Trek game. Two FS rules in the ATTACK table (top right) use the multi-output decision feature. They are indicated by the "&" symbol in the decision column.

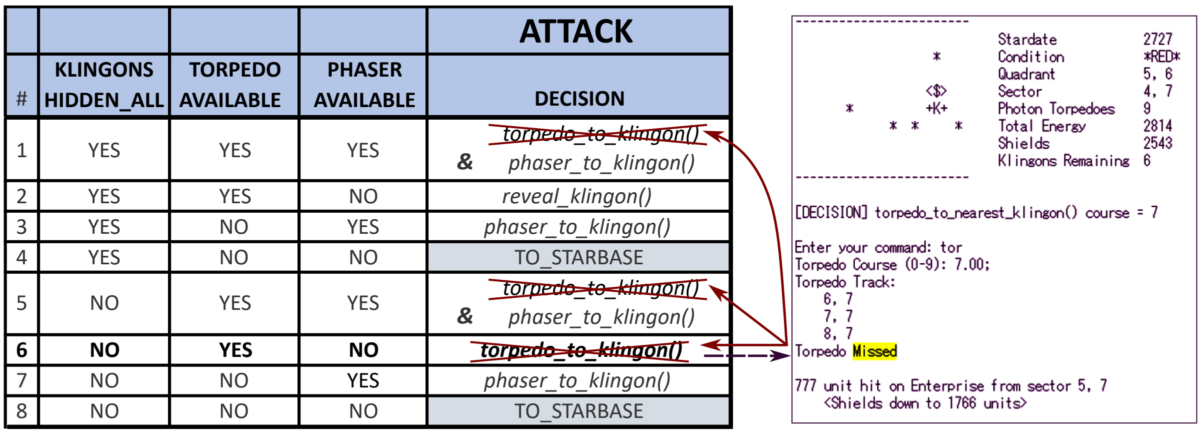
the decision making of Star Trek game. The output from one table may lead to an evaluation of another table. Each table represents a module that is responsible for one game action. In the figure, the decision in SHIELD table can lead to an ATTACK or a NAVIGATE when the SHIELD is already set.

4.4.2 FS Table Reevaluation

Occasionally, when playing a video game, it is possible for a game environment to make mistakes and give us wrong information. The FS decision-making system may compute a faulty decision due to an incorrectly inferred game state. In a worst-case scenario, when our player model keeps holding the same faulty decision, the game environment may stall due to no game progress.

To advance the game from the halt, we have to discard this faulty decision and re-evaluate the table without poor output decisions. The FS table re-evaluation activates automatically when a decision generates an invalid command feedback due to unexpected game events or wrong command arguments. In this case, the FS will be re-evaluated with the faulty decision disabled. This ensures us a new decision from the calculation.

To handle the repetitive case when the new decision is still faulty, we repeatedly disable it and re-evaluate the table until a good decision comes out. If all decisions in the table are faulty, we navigate up in the table hierarchy to disable the faulty output table then recalculate for the next working decision.



				ATTACK
#	**KLINGONS HIDDEN_ALL**	**TORPEDO AVAILABLE**	**PHASER AVAILABLE**	**DECISION**
1	YES	YES	YES	~~*torpedo_to_klingon()*~~ & *phaser_to_klingon()*
2	YES	YES	NO	*reveal_klingon()*
3	YES	NO	YES	*phaser_to_klingon()*
4	YES	NO	NO	TO_STARBASE
5	NO	YES	YES	~~*torpedo_to_klingon()*~~ & *phaser_to_klingon()*
6	**NO**	**YES**	**NO**	~~***torpedo_to_klingon()***~~
7	NO	NO	YES	*phaser_to_klingon()*
8	NO	NO	NO	TO_STARBASE

```
------------------------
                              Stardate             2727
                *             Condition            *RED*
                              Quadrant             5, 6
               <$>            Sector               4, 7
     *         +K+            Photon Torpedoes     9
          *  *    *           Total Energy         2814
                              Shields              2543
                              Klingons Remaining   6
------------------------

[DECISION] torpedo_to_nearest_klingon() course = 7

Enter your command: tor
Torpedo Course (0-9): 7.00;
Torpedo Track:
    6, 7
    7, 7
    8, 7
Torpedo Missed

777 unit hit on Enterprise from sector 5, 7
    <Shields down to 1766 units>
```

Figure 4.6: When a previous decision failed, FS re-evaluation is activated while the previous decision is disabled.

4.4.3 Multi-output decision

When designing FS rules for the FS decision-making system, it might be possible for a couple of decision outputs to be all suitable for a rule. Each of these options usually indicates different action behaviors. In this situation, a rule designer may face a difficult time to select, from these outputs, only one correct decision for the rule.

To assist a rule designer, we allow several output decisions to be expressed in an FS rule. We use an optimization process via evolutionary computation algorithm to, automatically, find out the appropriate option for the rule. Additional information on optimization of multi-output decision is explained in subsection 5.2.4.2.

4.5 Example Implementation of Game Player Model in Star Trek Game

4.5.1 The Star Trek Game Environment

We use a text-based TBS game to demonstrate our purposed methodology for automatic game parameter tuning. The benefit of using a text-based game is that its game output is done in a text format. This makes it quick and easy to parse text data and search for useful game states and information. It is, generally, challenging when playing a game with various strategies to work out.

Among a huge number of text-based TBS games worldwide, Star Trek is a simple well-designed one with merely eight game commands to use, as presented in Table A.1. Having been created since 1970s, the original text-based Star Trek game is still popular among the series supporters. The rules and mechanics of the game are well-accepted, and Star Trek has become an inspiration for many contemporary video games in the market. Considering these reasons including the availability of its source codes from the Internet [41], we select Star Trek game as our simulation test bed, representing a typical game in the turn-based strategy game genre. We present a brief introduction, detailed explanation of the gameplay, input commands, output displays, game maps, and some screenshots of Star Trek game in Appendix A.

When working with Star Trek game, we obtained its source code in C programming language. Therefore, we used C/C++ programming language in Visual Studio 2015 for 64-bit Windows 10 Education to develop a player model for the game. Though we kept the source code of Star Trek game intact for most parts, we replaced the pseudo-random number generator from C standard library, i.e. rand()

and srand(), with a more controllable pseudo-random number generator C++ class. We used the random number function originally proposed by Park and Miller [39] as the core algorithm. The code of this function is distributed with "Numerical Recipes in C" [43].

Pseudo-random number generator plays a major role to provide random objects or events in most of the computer software, especially in video games and EC algorithms. In Star Trek game, we use random numbers to position every game object in the galaxy, assign attacking energy for Klingon spaceships, determine a Klingon's move during a battle, specify damage values for the Enterprise's devices, etc. In a DE optimization algorithm, we use random numbers to initialize DE population, generate mutation vectors and trial vectors, facilitate crossover operation, etc.

To provide a fair simulation setup in which the games always do the same behaviors every time the same input is given within the same environment, we require exact controls over the random number generator. With random functions from standard C library, it is difficult to control the generated random numbers when simultaneously used with two applications: a DE algorithm and a video game. In addition, the random functions from standard C library lack the ability to pause, save, and resume the applications while maintaining the expected randomness. Hence, we have to replace the random functions with the ones that is more flexible for our use to save and resume the simulations. With the new random function, we implemented two ID numbers for a game and an DE optimization. The ID number acts like a random seed to make sure that we can recreate or rerun the same DE simulation setups on video games using the given ID numbers.

We also modified the source code in Star Trek to extend its input and output interface. We then created a connection between the interface of our player model and the one of the game. After that, we implemented C++ classes for each element of our player model structure, e.g. game data parser, decision-making module, game command generator, etc. as described in section 4.3.

4.5.2 The Interface Module for Star Trek Game

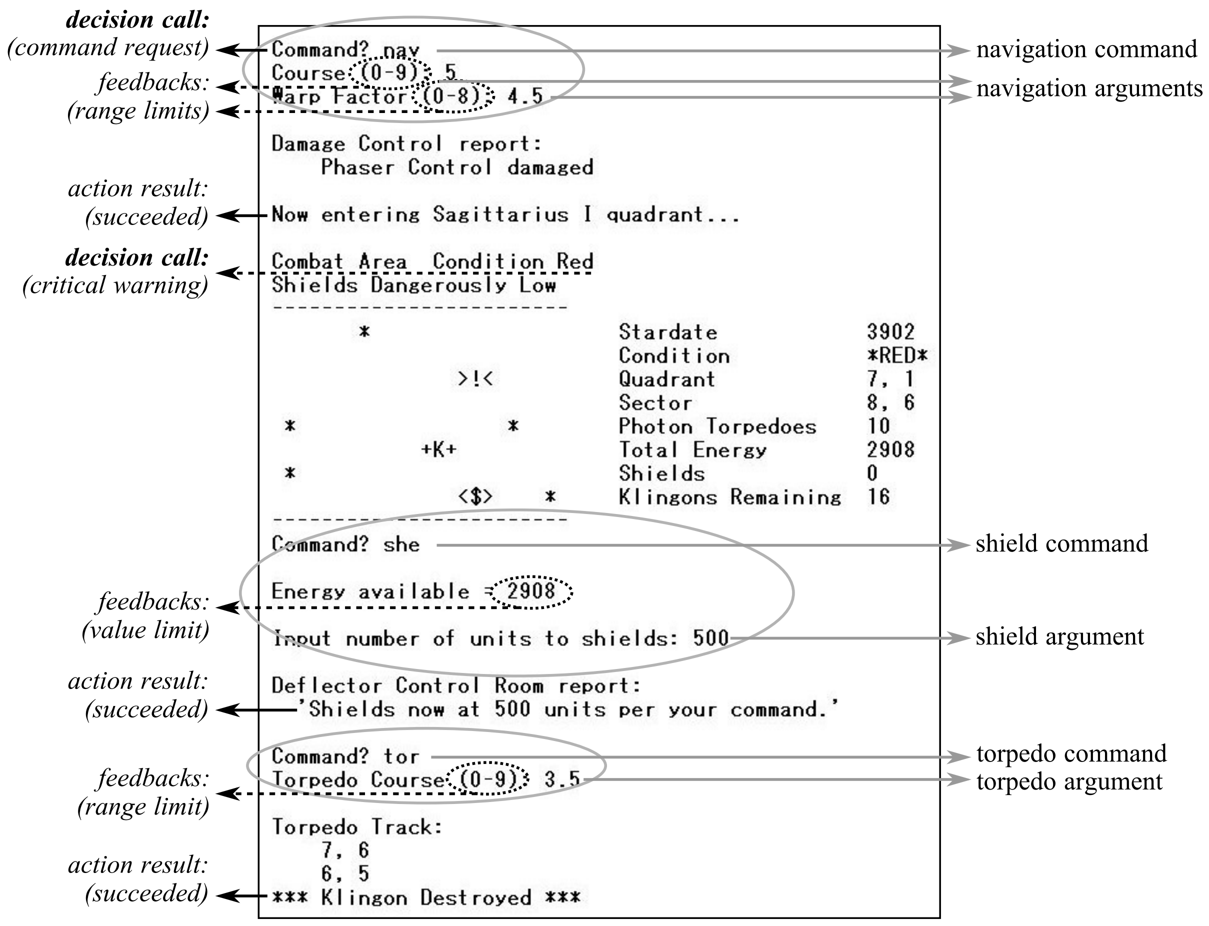
Our implementation of a player model for Star Trek game starts with the interface module which communicates with the game as the model input (game data output) and model output (game command input). We also need to provide our system an ability to read simulation data from input files as well as write simulation results and statistics to output files accordingly. Notable implementations for the interface module are:

- **Input text buffer & redirections:** We store game text data in the text buffer of the player model. Consequently, we are able to redirect the buffer to several input channels. The primary channels are (1) the standard display (stdout) to maintain the original output channel of game output, (2) the input to our game data parser (see subsection 4.3.1), and (3) the logfile system to record game data for human decision logs (see section 7.2) including game results and statistics.

- **Output text buffer & redirection:** Similar to the input buffer, we create an output text buffer to store the game commands of the player model as well as input characters of a human player. The primary output redirection channels are (1) the standard display (stdout) to maintain the original output channel of game input, (2) the output from our game command generator (see subsection 4.3.3), and (3) the logfile system to record game commands for human decision logs.

Figure 4.7 shows an example of a screenshot of Star Trek game. The left-hand side of the figure displays the game information produced by the game data parser. On the right-hand side, it displays the game commands and arguments supplied by the game command generator. At the same time, we can view this figure as interactions between the game environment on the left and the player model on the right, in terms of the game states.

Here is an interface scenario for the first command interactions in Fig. 4.7. Our game data parser searches for input texts with specific keywords to extract game information then classifies the information into three levels, as described in subsection 4.2.3. The "Command?" keyword specifies a call for a new decision. Our decision-making module then makes a decision to navigate. The game command generator creates an action plan and issues "nav" command back to the game. Next, Star Trek game sends out the command feedback asking for a command argument with specified range value. The command generator issues the command arguments according to the plan. The game then sends out the action results and a critical warning that the Enterprise now encounters the Klingons in a new quadrant. The data parser updates the new game states, retrieves the success result, and acquires a critical warning. This activates the decision-making module to make a new decision in response to a combat situation.



```
Command? nav
Course (0-9) 5
Warp Factor (0-8) 4.5

Damage Control report:
    Phaser Control damaged

Now entering Sagittarius I quadrant...

Combat Area  Condition Red
Shields Dangerously Low
------------------------
      *                 Stardate           3902
                        Condition          *RED*
             >!<        Quadrant           7, 1
                        Sector             8, 6
 *                *     Photon Torpedoes   10
          +K+           Total Energy       2908
 *                      Shields            0
             <$>    *   Klingons Remaining 16
------------------------
Command? she

Energy available = 2908

Input number of units to shields: 500

Deflector Control Room report:
   'Shields now at 500 units per your command.'

Command? tor
Torpedo Course (0-9) 3.5

Torpedo Track:
    7, 6
    6, 5
*** Klingon Destroyed ***
```

Figure 4.7: Screenshot of Star Trek game showing (left arrows) game information and (right arrows) game commands & arguments.

4.5.3 FS Rules for Decisions in Star Trek Game

FS rules for game decision are the most crucial part of the decision-making module. The authors created FS rules for Star Trek game after playing the game several times and working on different game parameter settings. We applied the modular FS table technique to create several small hierarchical FS tables. We then came up with two sets of FS rules as follows:

4.5.3.1 Simple FS Rules

In the simple rules, we have five modular tables made of five binary FS inputs (LOW or HIGH) and nine Boolean inputs. Appendix B shows all five binary inputs and five FS tables in section B.1 and B.2, respectively.

Those five FS tables in the simple rules are:

1. **SHIELD table:** SHIELD table, presented in Table B.2 in Appendix B, serves as the root of the modular tables. The first priority of this table is to control and balance the shield energy of the Starship Enterprise. Without a proper shield setting, it is most likely that the Enterprise will be defeated by the Klingons and, instantly, we lose the game. The shield setting is an output decision which points to the decision callback function named *set_shield_energy()*. This C function is prompted to calculate a proper amount of shield energy, suitable for the current game states.

   Besides setting the shield energy, this table may point to one of three other FS tables under the hierarchy of the root. When there is a Klingon spaceship within the same quadrant, ATTACK table is next activated. When there is a Starbase within the same quadrant and we need to repair our starship, we move to TO_STARBASE table to further find a way to the Starbase. If there are both Klingon and Starbase, we can choose to *set_shield_energy()* higher, make an ATTACK, or move TO_STARBASE, depending on some other factors specified in the table. Otherwise, when there is nothing else to do in the current quadrant, we follow to NAVIGATE table then find out a new quadrant to move on.

2. **ATTACK table:** The ATTACK table, presented in Table B.3, is responsible for selecting appropriate weapons to attack the Klingons. The callback function *fire_phaser()* calculates the energy to discharge the phaser, while *fire_torpedo()* computes the direction to launch the torpedo.

   If a torpedo is the right weapon but there exists a star, blocking the target Klingon, we may decide to *reveal_klingon()* first to make a clear path before firing a torpedo in the following turn. Otherwise, when all weapons are unable to fire, we escape TO_STARBASE to repair the damage or recharge the weapons.

3. **NAVIGATE table:** The NAVIGATE table, presented in Table B.4, guides the navigating decision toward TO_STARBASE or TO_KLINGON table. The weapon availability as well as the remaining energy and time are three factors that dictate the decision.

4. **TO_STARBASE table:** This table, presented in Table B.5, determines if we should directly navigate to the known Starbase or try to survey the undiscovered quadrants in search of the unknown Starbase. Our experience would advise us to *navigate_to_starbase()* when it is near. Otherwise, we should *navigate_to_survey()* to find out more in the galaxy.

5. **TO_KLINGON table:** Table B.6, the TO_KLINGON table, is a simple table with a yes/no condition. We decide to *navigate_to_klingon()* when we know its location. Otherwise, we have to *navigate_to_survey()* to search for it in the galaxy. The callback function *navigate_to_klingon()* determines the nearest quadrant where the Klingons exists. Whereas the *navigate_to_survey()* callback directs us to the quadrant where we have as much opportunity as possible to discover the unknown galaxy.

4.5.3.2 Extended FS Rules

We create an extended version of the simple FS rules by adding more rule details and introducing FS value tables. In addition to the modular FS tables for decision making, we implement an FS value table to calculate a command argument based on the relationship among FS input variables. Appendix C shows all 14 FS input variables and 8 FS tables for the extended FS rules. Among 14 FS variables, 10 of them belong to 6 modular tables for decision making, while 4 variables belong to 2 value tables for command argument calculation. In addition, there are six extra output variables shown in the bottom half of Table. C.1. These variables represent the outputs in the value tables, VALUE_SHIELD and VALUE_PHASER for shield and phaser command's arguments, respectively.

In the extended rules, we also employ two separate branches of modular tables. Starting with *MAIN_DECISION* table, the first branch is the main part similar to the one in the simple FS tables. The second branch is a critical decision that activates only when the Enterprise enter a new quadrant with existing Klingons. The critical decision for the extended rules contains only one table that is the *RED_ALERT* table shown in Table C.2.

All six FS tables for decision making in the extended rules are:

1. **RED_ALERT table:** This table is similar to SHIELD table in the simple FS rules in terms of the priority target: setting shield energy properly. However, we set it as a critical decision in a way that it can interrupt the regular decision

when a critical event occurs. Another point of differences is the number of DOM states in *SHIELD_ENERGY* input. With higher number of DOM states, we can adjust the output decision to match the input circumstances more precisely.

2. **MAIN_DECISION table:** This table splits major game events into four cases, all with Boolean condition. All table outputs point straightforwardly to one of the other three tables in the hierarchy.

3. **ATTACK table:** We add one more FS input variable in the simple rules' ATTACK table. The new input, *N_KLINGONS_IN_QUADRANT*, represents the number of Klingons in the quadrant: either LOW or HIGH. With HIGH number of Klingons, it is more effective to *fire_phaser()* to destroy all of them at once.

   However, we must use *fire_phaser()* sparingly to reserve the valuable energy. Thus, it is economical to *fire_torpedo()* when the number of Klingons are LOW.

4. **NAVIGATE table:** We add a new factor *DISTANCE_TO_STARBASE* to consider the *TO_STARBASE* decision for an energy recharge when a Starbase is near. This often prevents the Enterprise from running out of energy.

   We also replace the input variable *WEAPON_AVAILABLE* with *DAMAGING DEVICES* to emphasize the need to repair the damage with *TO_STARBASE* decision.

5. **TO_STARBASE table:** The TO_STARBASE table in the extended rules is the same as the one in the simple rules.

6. **TO_KLINGON table:** We add time factor as *TIME LEFT* input variable to give more weight on the galaxy survey when we have more time to do so. When the time is running out, we shift the weight to *navigate_to_klingon()* to win the game.

The other two tables in the extended rules are FS value tables. The FS value table calculates an output value based on the relationship between FS inputs and the constant consequence values.

When we use the simple FS rules with the optimization process in Chapter 5, we assign some optimized parameters to command arguments directly. However, these acquired values are fixed for the entire simulation. Besides, they have no association with the current game states in the interactive environment. This direct use of optimized parameters is inflexible when compared to a new use of optimized parameters in an FS value table.

In an FS value table, we assign the optimized parameters to the consequences of the table. The consequences specify different amount of output values for differing input conditions. We then use FS reasoning to interpolate these table consequences in association with the current game states. That means, when the current game states are given as table inputs, we calculate the membership values for each input and compute the weight for each rule according to the relationship of the inputs. We then multiply this rule weight to the consequence values, resulting to the weighted consequence. The accumulation of all weighted consequences specifies the output values of the FS value table. This provides a more flexible way to use the optimized parameters in the varying environment.

When a callback function determines an argument value for its command, the function may activate the FS value table to calculate an output value as the required command argument.

Both FS value tables in the extended FS rules are (with continue numbering):

1. **VALUE_SHIELD table:** The callback function *set_shield_energy()* activates VALUE_SHIELD table to calculate the proper shield energy for "she" command. The command sets shield energy of the Enterprise to protect this starship from the Klingons attack.

   We set shield energy in proportion to the strength of Klingons attack, which depends on the number of Klingons in the quadrant and the distance from each Klingon to the Enterprise. The more the number of Klingons is, the more shield energy is required to defend the starship. Meanwhile, the closer the Klingon locates near the starship, the more powerful the attack becomes. For both cases, we need more shield energy to protect the Enterprise.

2. **VALUE_PHASER table:** The callback function *fire_phaser()* activates VALUE_PHASER table to calculate the required phaser energy for "pha" command. The command fires the phaser energy to attack the Klingons.

   Firing the phaser weapon to destroy faraway Klingons needs more phaser energy than to defeat the ones nearby. When the number of Klingons increases, we also need more phaser energy to eliminate all Klingons in one blaze. Therefore, the effective amount of phaser energy depends on the number and the distance of Klingons in the quadrant.

4.5.4 FS Membership Functions

For FS membership function, we implement a triangular-shaped membership function for its quick computation and the minimum number of function parameters.

We use a series of triangular functions, enclosed with a boundary function at both ends, to represent membership functions of an FS variable. The number of functions equal to the number of DOM states. The shoulder function defines the boundary in which all other triangular functions reside.

The main characteristics of a membership function in our implementation are the unit height and the unit summation. That is the height of a triangular membership function equals to one. At any position along the x-axis, the summation of the overlapped membership functions always equals to one.

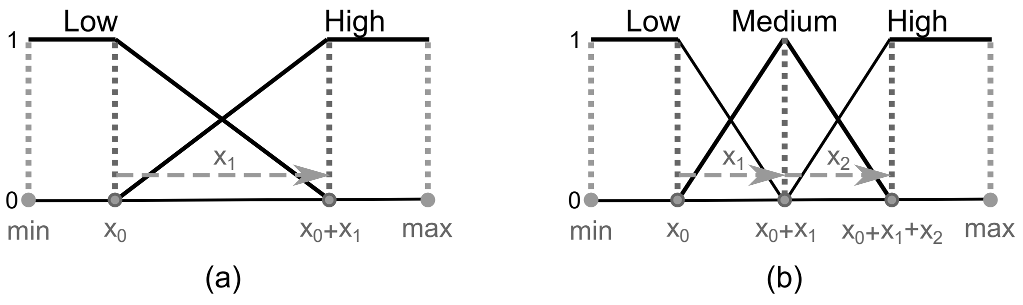


Figure 4.8: Membership functions of an FS variable with (a) two degrees of membership (Low & High) and (b) three degrees of membership (Low, Medium & High).

Figure 4.8 shows our implementation of triangular membership functions used in a player model for Star Trek game. Initially, we indicate a minimum and maximum values of each FS variable. Both values specify the boundary of the game state which an FS variable denotes to. All valid membership functions reside within this boundary range. Any input value outside this range is not a member of the FS variable. Its membership value is always zero.

In addition to a boundary range, an FS variable with n DOM states requires n real values to create its membership functions. Fig. 4.8 (b) demonstrates that we need three real numbers to represent membership functions with three DOM states: Low, Medium and High. The lower number establishes the left shoulder function that maintains unit height from the minimum boundary to its position then linearly decreases the height from its position down to zero height at the position of the higher number. Similarly, in the opposite way, the higher number establishes the right shoulder function that the height linearly increases from zero at the position of the lower number to one at its position then maintain the unit height until the maximum boundary. We insert additional triangular functions between both shoulder functions until the total number of functions equals to the number of DOM states.

4.5.5 Game Decision Callbacks

In our implementation, the decision output from FS tables determines the game decision for a player model. This decision gives the generic direction for the game response, without action details. For each FS decision outputs, we implement a decision callback function to serve as an action planner and a command generator. The player model activates the callback function that corresponds to the given game decision.

In Star Trek game, we design seven callback functions according to all main decisions in the game. According to the given FS tables in Appendix B.2 and C.2, the name of the callback functions appears in the DECISIONS column as lower-case letters with a parenthesis ending. Table 4.1 lists all callback functions along with their matching game command and arguments.

Table 4.1: Star Trek game's callback functions to generate the corresponding game command with its arguments.

name	command	argument#1	argument#2
set_shield_energy()	"she"	energy	-
fire_phaser()	"pha"	energy	-
fire_torpedo()	"tor"	Course (direction)	-
reveal_klingon()	"nav"	Course (direction)	Warp Factor (distance)
navigate_to_survey()	"nav"	Course (direction)	Warp Factor (distance)
navigate_to_starbase()	"nav"	Course (direction)	Warp Factor (distance)
navigate_to_klingons()	"nav"	Course (direction)	Warp Factor (distance)

All seven callback functions and their detailed operations are:

1. **set_shield_energy()**: The main task of this callback function is to issue the "she" command with an appropriate amount of shield energy.

   The importance of shield energy is to protect the Enterprise from being destroyed. If the shield energy is too low, we may lose the game due to a more powerful Klingons attack.

In the optimized player model proposed in Chapter 5, we let the optimization process search for the proper shield energy. For the simple FS rules, we assign the optimized parameter to the shield value directly. However, for the extended FS rules, we activate the VALUE_SHIELD table to compute this value every time we issue the “she” command. The VALUE_SHIELD table is shown in Table C.8 of Appendix C.

2. **fire_phaser()**: The main task for this callback function is to issue the “pha” command with a proper amount of phaser energy.

   The balance of energy usage is the key to specify the right energy value. When firing a phaser weapon, we aim to destroy all Klingons in one shot, if possible. Otherwise, the surviving Klingons will fight back. We also need to avoid firing more phaser energy than necessary. It wastes our precious energy, then we undoubtedly lose the game once the energy is used up.

   Similar to the shield energy, the callback assigns the optimized parameter to the phaser energy in the simple FS rules. For the extended FS rules, it activates the VALUE_PHASER table to calculate the suitable phaser energy. Table C.9 of Appendix C shows the relationship between related game states and the proper amount of phaser energy.

3. **fire_torpedo()**: The main task for this callback function is to issue the “tor” command. The function has to determine the torpedo direction to suit the required command argument.

   We calculate the torpedo direction algorithmically from the quadrant map. The query command “srs” shows the map of all 8×8 sectors in the current quadrant.

   A star object in the quadrant can block the path of our torpedo direction. In such case, we can move the Enterprise to make a clear path toward our launching target.

4. **reveal_klingon()**: The main task of this callback function is to issue “nav” command(s) to relocate the Enterprise for a clear path toward the Klingons. This command focuses on a short move within the same quadrant.

We usually issue this command in a preparation for a torpedo launch. After this move, it is the Klingons' turn and they can attack the Enterprise before we launch a torpedo. This is the drawback of *reveal_klingon()* decision. Therefore, we make this decision only when there are no better choices.

The relocation for a clear path requires small-step navigations or a series of them. It depends on the distribution of the objects in the quadrant: stars, the Klingons, and the Enterprise. In addition, the Klingons relocate occasionally. Hence, a poor relocation that cannot clear the path in one move is dangerous.

In *reveal_klingon()* function, we use the brute-force method to search for the new locations with a clear path to the Klingons. With the small number of 64 sectors in a quadrant, the brute-force method works gracefully.

In the implementation, we initially find all vacant locations that the Enterprise can move to, positively, from the current position, without any obstructions. We rate them in terms of accessibility and clarity of the travelling path from the Enterprise's position. We then rate these locations again for the clarity of torpedo path to the target object. The best rated location is our candidate for the "nav" command.

We implement Bresenham's line algorithm [2] to trace a straight line between two sectors in a quadrant. We use this algorithm to check whether there are no objects along the line to ensure the clarity of a travelling path.

5. **navigate_to_starbase()**: The main task of this callback function is to issue "nav" command(s) to dock the Enterprise into the Starbase. The docking positions are any eight sectors immediately surrounding the Starbase. This command handles both a long move between quadrants as well as a short move within a quadrant.

   For a long move between quadrants, this function creates a series of "nav" commands starting from the current position to the target quadrant. The query command "com 0" displays the galaxy map, as illustrated in Fig. A.2 in Appendix A. We use this galaxy map to plan the moving actions. We also need to be concerned about the obstruction of star objects in the current quadrant when planning the moves.

   For a short move inside a quadrant, we also use the brute-force method to search for the candidate travelling paths to the neighboring locations of the Starbase.

6. **navigate_to_survey()**: The main task of this callback function is to issue "nav" command(s) to move the Enterprise when surveying the galaxy. This command focuses on a long move between quadrants.

   The crucial objective of galaxy survey is to search for all Klingons in the galaxy. The query command "lrs" reveals the numbers of each object residing in the current quadrant and its immediate surroundings. From point to point, we use "lrs" and "nav" command to gradually survey the galaxy.

   With the brute-force method, we search for the quadrants that contain a good balance of our survey measures. Currently, we use two measures in the ranking: (1) the travelling distance from the Enterprise to the target quadrant to be surveyed, (2) the number of newly discovered quadrants we can scan from the target quadrant. We also apply the weight system to adjust the importance of each measure. The quadrant with top-ranking weighted measure is our survey target for the navigation.

7. **navigate_to_klingons()**: The main task of this callback function is to issue "nav" command(s) to move the Enterprise to the quadrant where the Klingons are located. This command focuses on a long move between quadrants.

   We select the target quadrant to fight the Klingons in the similar way that we measure the survey ranking. From the galaxy map, we apply two weighted measures, i.e. the travelling distance and the number of new discoveries, to the quadrants where the Klingons reside. The top-rank quadrant is the destination quadrant of the navigation.

4.6 Chapter Summary

Our game player model is created after the model of how a game player interacts with a game environment. For a strategy game, it is the decision-making process that dominates the interaction.

In this work, the player model consists of three major modules: (1) the game parser module as an input interface, (2) the decision-making module, and (3) the command generator module as an output interface. We use Star Trek game to demonstrate the approach in an actual game environment.

The capability of a player model for strategy games lies in the competency of the underlying decision-making techniques. Selecting the suitable decision-making algorithms depends on the complexity of strategy decisions in the game. For TBS games, we apply the FS rule-based system to make the game decisions. With the expressive power of FS rule-based system, an expert player can linguistically communicate their knowledge in game playing into IF-THEN rules. This helps the non-technical professionals design their game decisions in human terms. In addition, the FS rule-based system is easy to understand and maintain by others not originally creating the system.

The following chapters exhibit various aspects of our implementation of the player model for TBS games. That includes several improvements of the model to make it stronger as well as some discussions on the evaluation and other enhancements.

Chapter 5

Evolving Fuzzy Logic Rule-based Player Model

5.1 Introduction

After an expert creates a player model representing a game player as described in Chapter 4, the first step toward automatic game parameter tuning is to optimize the player model to become a stronger automatic game player. This reduces the workload of a game designer in fine-tuning the player model for the optimal performance. With fuzzy logic (FS) decision system in our player model, we can use an evolutionary algorithm (EA) to optimize the FS decisions for more robust model instances. This technique is called genetic fuzzy system (GFS) [55].

Genetic Fuzzy Systems, basically, apply evolutionary algorithms (EAs) to solve optimization problems and search problems related to FS, especially fuzzy rule-based system (FRBS). For a FRBS, we analyze its fuzzy knowledge base (KB) component as an optimization problem and use an EA as a learning tool in the design of KB component. Examples of GFS techniques include genetic rule learning, genetic rule selection, genetic tuning of membership function parameters, etc. [20].

It is worth to mention that GFSs are different from fuzzy evolutionary algorithms (fuzzy EAs), the other method in the hybridization between FS and EAs [20]. In fuzzy EAs, we improve EA performance by using FS to control or model EA elements adaptively.

The main objective of this chapter is to confirm that we can use a GFS technique to improve the performance of our player model for turn-based strategy games at different levels of game difficulty.

Following this section, the remaining three main sections discuss the player model optimization with Differential Evolution (DE) algorithm and its implementation, the simulation experiments with its results, and the chapter summary in section 5.2, 5.3

and 5.4, respectively.

5.2 Player Model Optimization with Evolutionary Computation

The main element of our player model is an FS decision system, which is responsible for decision makings in the game. This decision system is the major target for player model optimization with GFS technique. Although we use DE in our experiments, we are still able to use any EC algorithm for the optimization without significant differences [13].

In Fig. 5.1, we show a traditional practice of an FS system design. In this practice, an expert uses his/her skills in the domain to create FS rules. The FS membership function parameters are also tuned by the domain expert him/herself.

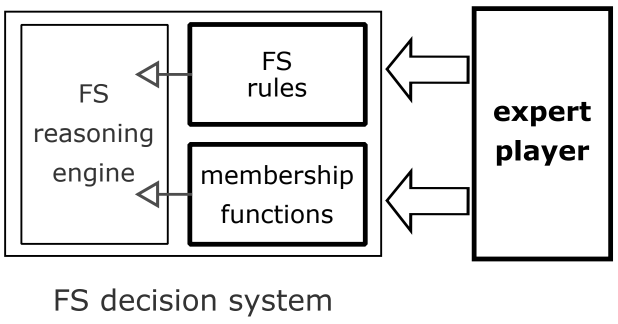


Figure 5.1: An expert player provides both FS rules and membership function parameters for the game decision.

To help automate the tuning process of FS membership function parameters, our framework uses an EC algorithm to search for the optimal values of the membership function parameters. as illustrated in Fig. 5.2.

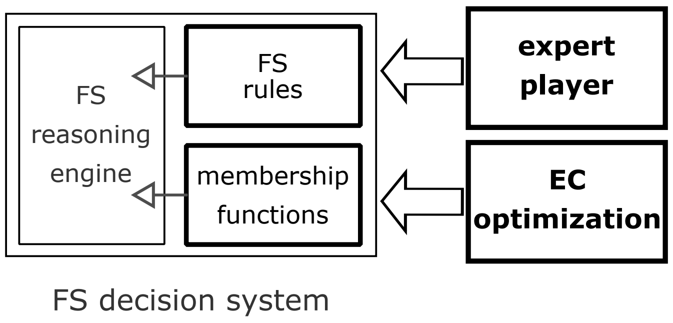


Figure 5.2: An expert player provides FS rules, while DE optimizes membership functions for the optimal function parameters.

Although both FS rules and membership function parameters can be a subject of optimization or automation, we prefer to leave the creation of FS rules in the hands of experts. We believe that FS rules are well analyzed and synthesized from their skills and experiences. The rules created by experts, generally, contain many insights and intuitions, which are difficult to obtain from the algorithms, or may take a long time to search for from the vast search space.

Once FS rules have been created, finding the optimal values of membership function parameters seems to be more reasonable for a search algorithm due to the smaller search space and less complicated tasks. In addition, the optimization of membership function parameters requires precise and extremely detailed tweakings, which are better handled by an optimization algorithm.

EC optimization applies a population-based EC algorithm to search for the optimal fitness values over the search space. The search space for solutions to the problems is in a form of encoding representation. The encoding maps the problems into the parameter space in which we search for the answers. Applying fitness evaluation over the search space constitutes the fitness landscape of the problems. In addition to an efficient EC searching algorithm, a well-represented encoding and a faithful fitness evaluation facilitates the searching/optimization process.

There are many factors concerning the optimization of EC algorithms. The three most important ones are (1) *the algorithm used*, (2) *the fitness evaluation*, and (3) *the encoding representation*. We will discuss these three issues in details in subsections 5.2.1, 5.2.2, and 5.2.3, respectively. In subsection 5.2.4, we explain how to evolve our player model to find a stronger one with EC optimization. We complete the section with an implementation example of player model optimization for Star Trek game in subsection 5.2.5.

5.2.1 Differential Evolution Algorithm

Generally, we can use any population-based EC algorithms for the optimization process. In this case, we select a DE algorithm, described in subsection 2.1.4, as our optimization tool. The main advantages of DE are its simplicity and usability. Moreover, DE is efficient for optimizing real-valued, multi-modal problems. It yields good convergence properties and performs well at avoiding local optima.

Similar to other EC algorithms, DE searches for the optimal solution in the fitness landscape, represented by encoding representation and fitness evaluation through three evolution processes: mutation, crossover, and selection. We control the mutation process with the selected DE notation (see Table 2.1) and the scaling factor F, as shown in Eq.(2.3). A large value of F indicates a wide-area searching (exploration), while a small value indicates a local-area searching (exploitation). We manipulate the crossover process with the crossover rate CR, as specified in Eq.(2.4). A large value of CR indicates a high crossover rate, implying distant searches far away from its parent (exploration), while a small value suggests nearby searches (exploitation).

5.2.2 Fitness Evaluation

Fitness evaluation is one of the most important issues in the design of an EC algorithm for a specific application. Each application requires different fitness evaluations depending on the objectives of the EC optimization. In our optimization, the main objective is to make the FS decision system in our player model as strong as the human game player.

Hence, we design our fitness evaluation toward winning the games. That is to win as many games as possible or, if unavoidable, to lose as few games as possible.

To evaluate a player model, we arrange the model to play the game many times with, for each time, distinctive game characteristics, i.e. different game maps, different seed values of pseudo-random number generator. This is to, statistically, make a fair measure on the performance of the player model.

$$S_{fitness}(g, m, N) = \frac{\sum_{id=1}^{N} S_{game}(g, m, id)}{N} + R_{win}(g, m, N) \tag{5.1}$$

Where:

g:	is a set of game parameters,
m:	is a set of player model parameters,
N:	is the number of played games,
$S_{fitness}$:	is a fitness score,
S_{game}:	is a game score according to Eq. 5.2,
R_{win}:	is a win ratio according to Eq. 5.3.

We express our fitness evaluation in Eq.(5.1). A stronger player model has given a higher numerical value than a weaker model has. The equation consists of two terms:

1. **An average of game scores**: This value is an arithmetic mean of game scores. Generally, game scores represent how well we perform in a single game. A basic form of possible game scores is a weighted sum of significant

game statistics. To make it comparable, we normalize the game statistics as well as the overall game scores. Eq.(5.2) shows the mathematical equation of this conceptual score.

$$S_{game}(g, m, id) = \frac{\sum_{i=1}^{M} w_i \times S_i^{Norm}}{\sum_{i=1}^{M} w_i} \tag{5.2}$$

Where:

g:	is a set of game parameters,
m:	is a set of player model parameters,
id:	is a game ID number,
M:	is the number of game statistics of optimization importance.
S_i^{Norm}:	is the i^{th} normalized game statistics.
w_i:	is the i^{th} weight of corresponding game statistics.

2. **A winning ratio**: This value represents the overall performance of the player model. The winning ratio, expressed in Eq.(5.3), is a motivation toward winning more games.

$$R_{win}(g, m, N) = \frac{\sum_{id=1}^{N} W(g, m, id)}{N} \tag{5.3}$$

Where:

g:	is a set of game parameters,
m:	is a set of player model parameters,
N:	is the number of played games,
$W(g, m, i)$:	$= 1$ when win the game of ID:i
:	$= 0$ otherwise.

Each term is normalized to the value between zero and one. Thus, the comparison between two player models with different numbers of games played is reasonable.

We show our implementation example of fitness evaluation function in Star Trek game in subsection 5.2.5.

5.2.3 Encoding Representation

To improve our player model with EC optimization, the video game serves as a problem that we are trying to solve with the suitable player models. The behaviors of the player models are defined by player model parameters, which are the encoding representation of the solutions to the problem. A good design of player model parameters also supports the optimization process.

The total number of parameter size is important. The size of player model parameters should be large enough to finely represent the problems. Too few parameters to capture the characteristics of the problem may lead to impractical results. Too many parameters, however, take too much time to optimize or, in a worst-case scenario, make it impossible to find any optimal solutions. We must balance the size of player model parameters to closely represent the problems as well as to search for the solutions within a preferable time frame.

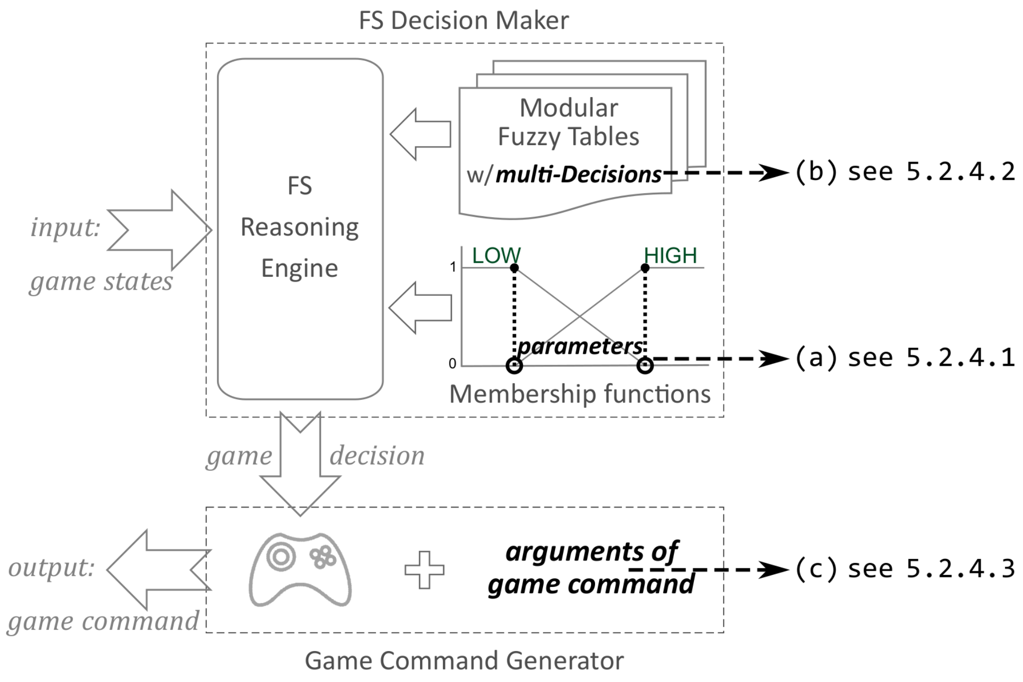


Figure 5.3: player model parameters.

For our proposed methodology, we select three key parameters from our player model. These key player model parameters are illustrated in Fig. 5.3 as (a) *the membership function parameters*, (b) *multi-output decisions* , and (c) *arguments of game commands*. Two parameters belong to the KB component in the decision-making module. The membership function parameters are from FS membership functions, and multi-output decisions are from FS rules. The other parameter belongs to the game command generator module. They are the target of our DE optimization. In the next subsection, we will discuss the optimization of each parameter in more details.

We show our implementation example of encoding representation in Star Trek game in subsection 5.2.5.

5.2.4 Optimization of Player Model Parameters

The optimization process produces sets of player model parameters from DE algorithms. One set of player model parameters specifies one player model. The player model makes decisions in games according to its parameters. It generates all game commands and creates game actions. When these actions interact with the video game environment, a variety of game states emerge. At the end of the game, we select some game statistics and evaluate the fitness scores for the given set of player model parameters. DE algorithm uses sets of parameters with high fitness scores to reproduce new sets of parameters. We repeat the selection and reproduction procedures until the set of optimal player model parameters, denoting the best game player, is found.

5.2.4.1 Optimization of FS Membership Function Parameters

During the inference process in FS decision making, an FS membership function maps an FS input variable and its corresponding DOM to a numerical value. The shape of a membership function and its DOM computation are controlled by the membership function parameters. With different sets of membership function parameters, various numerical interpretations of the FS input can be achieved, resulting to a variety of game decisions.

A large number of player model parameters belong to FS membership function parameters. The size of membership function parameters is proportional to the number of DOM states in an FS variable that the function represents. In addition, its size also depends on the shape of the membership function. Different types of function shapes require different numbers of the function parameters. For example, the parameter size of a simple triangular-shaped function is smaller than the size of a trapezoid-shaped.

5.2.4.2 Optimization of FS Multi-output Decision

To implement the multi-output optimization, DE algorithm assigns a weight for each multi-output decision. When calculating the weight of a rule with multiple outputs, we proportionally share the calculated rule weight according to the weight of each decision. We then accumulate the shared weights into the output weight of the corresponding decision. Finally, the output with the maximum accumulated weight is selected as the final decision. With our implementation of modular FS table feature, this final decision may point to another FS table for further reasoning computations or to a decision callback for action planning and game command generation.

The number of gene elements, required for multi-output optimization, is proportional to the number of FS rules, containing multiple outputs, and the number of output choices, provided in such rules. The number of encoding representations for multi-output optimization tends to be small when compared to the one of other types of representation. This is because only a few rules require this supplementary feature even though it is very handy in necessary cases.

In a representation of a gene element, we encode the multi-output decision with the focus on maintaining the unity weight distribution among multiple outputs. The number of gene elements we required is one degree of freedom less than the number of output choices in the rule. Assuming a rule with three possible decisions, we require two gene elements to encode their weights. The last weight is calculated from the first two values to satisfy the unity of their total weights. In the same example of the three optional decisions, suppose we have two encoding elements of 0.4 and 0.5, respectively. The weight of the first decision is straightly 0.4. The weight of the second decision is 0.5 of the rest, i.e. $0.5 \times (1.0 - 0.4)$, that equals to 0.3. To maintain the unity of total weights, the remaining weight, i.e. $1.0 - (0.4 + 0.3) = 0.3$, belongs to the last decision. Hence, the two encoding values of $(0.4, 0.5)$ are equivalent to $(0.4, 0.3, 0.3)$ of three-output weights. The summation of the three weights is one, which conserves the unity weight distribution we focus on.

5.2.4.3 Optimization of Game Command Arguments

A command argument(s) is required in some game commands as supplementary data to enhance the functionality of the commanding action. An argument of a game command is normally in a numerical form, but some other forms, e.g. a character or a symbol, are also possible. There are times when it is difficult to indicate the command argument only with FS rules. In this case, we have to supply the command argument with some other methods such as a constant value, a random value, a computed value, or an optimized value.

In command argument optimization, we assign a value of a specific gene element for a specific command throughout the game. The optimized command argument is used in either one of the following ways.

- Preset command argument: We use the optimized value directly as a command argument. This is usually for a simple game command.

- Callback function argument: We use the optimized value as a function parameter of a decision callback. The callback function calculates an actual value for the command argument. This is an indirect use of the optimized value, usually for a more complex game command.

An example of our implementation in command argument optimization for Star Trek game is presented in subsection 5.2.5.

The number of gene elements, required for game command arguments, depends largely on the input commands to the game environment, as well as the design of the KB component in our player model. The game command generator module acquires prospect command arguments, then generates the actual game commands according to the given decision. The prospect command arguments come from several methods, depending on our KB design for each specific game. The possible methods are static values from the predefined constants or random number generator, adaptive values from decision callbacks or optimized gene elements, or the mixture of both. Hence, the gene number varies drastically among game commands and game developers.

5.2.5 Example Implementation of Player Model Optimization in Star Trek Game

In the experiment for this chapter, we use simple FS rules (see section B) in the KB component of the player model. From all three kinds of key parameters, the simple FS rules contain 21 player model parameters in total. This is illustrated in Fig. 5.4 (a)-(c).

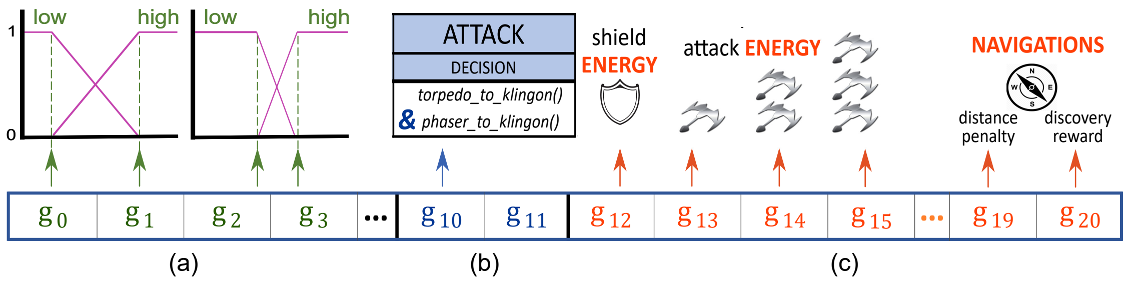


Figure 5.4: DE optimization for player model.

5.2.5.1 Parameter Encoding for DE Optimization

Table 5.1 shows our encoding representation of all 21 player model parameters. They consist of ten FS membership function parameters (gene $g_0 - g_9$), two multi-output decision parameters (gene $g_10 - g_11$), and nine game command arguments (gene $g_12 - g_20$).

Table 5.1: DE encoding representation for simple FS rules in Star Trek game. Three types of our player model parameters in *type* column consists of FS membership function parameters (*MFn*), multi-output decision (*Outs*), and game command arguments (*Cmd*), where *Cmd1* denotes a preset command argument and *Cmd2* denotes a callback function argument.

gene	min	max	type	representation
g_0	0.0	3000.0	MFn	LOW membership of SHIELD_ENERGY
g_1	0.0	3000.0	MFn	HIGH membership of SHIELD_ENERGY
g_2	0.0	3000.0	MFn	LOW membership of ENERGY_LEFT
g_3	0.0	3000.0	MFn	HIGH membership of ENERGY_LEFT
g_4	0.0	100.0	MFn	LOW membership of TIME_LEFT
g_5	0.0	100.0	MFn	HIGH membership of TIME_LEFT
g_6	0.0	5.0	MFn	LOW membership of UNKNOWN_STARBASE
g_7	0.0	5.0	MFn	HIGH membership of UNKNOWN_STARBASE
g_8	0.0	8.0	MFn	LOW membership of DISTANCE_TO_STARBASE
g_9	0.0	8.0	MFn	HIGH membership of DISTANCE_TO_STARBASE
g_{10}	0.0	1.0	Outs	Output move_to_show_klingon() from rule no.1 ATTACK table (see section B.3)
g_{11}	0.0	1.0	Outs	Output fire_torpedo() from rule no.5 ATTACK table (see section B.3)
g_{12}	0.0	3000.0	Cmd1	argument of SHE command
g_{13}	10.0	1000.0	Cmd1	argument of PHA command when fighting 1 Klingons
g_{14}	10.0	1000.0	Cmd1	argument of PHA command when fighting 2 Klingons
g_{15}	10.0	1000.0	Cmd1	argument of PHA command when fighting 3 Klingons
g_{16}	1.0	8.0	Cmd2	radius factor to calculate arguments of NAV command when exploring the galaxy
g_{17}	-1.0	1.0	Cmd2	distance weight to calculate arguments of NAV command when exploring the galaxy
g_{18}	-1.0	1.0	Cmd2	new exploration weight to calculate arguments of NAV command when exploring the galaxy
g_{19}	-1.0	1.0	Cmd2	distance weight to calculate arguments of NAV command when travel to fight the Klingon
g_{20}	-1.0	1.0	Cmd2	new exploration weight to calculate arguments of NAV command when travel to fight the Klingon

We implement FS membership function with a triangular function as illustrated in Fig. 5.4 (a). Thus, each DOM state requires only one function parameter. According to the simple FS rules in section B.1, all five FS input variables are two-state DOM: LOW and HIGH. Hence, we use ten gene elements in this part, i.e. two gene elements for each FS input variable.

Our encoding contains two gene elements for multi-output decision, the least number of gene elements among all three types of key parameters. This is because, out of 42 rules in 5 tables, only two rules in the ATTACK table implement this feature, both with two-output decision. As we need one degree of freedom less than the number of output options for the encoding of the feature, one gene element is used for each two-output rule. Hence, we use two gene elements in this part.

For game command arguments, there is no exact formula to compute the number of gene elements. It varies greatly due to our design of game command generator module. In our simple player model parameters, we use preset command arguments for SHE and PHA commands, and callback function arguments for NAV command. For SHE command, the value of g_{12} is issued as an expected shield energy. For PHA command, where the efficiency of the phaser attack depends mainly on the number of Klingons in the quadrant, we assign three gene elements for different numbers of Klingons. For example, when there are two Klingons in the quadrant, the player model will fire the phaser with an energy of g_{14}. For NAV command, we categorize the navigating actions into two groups: NAV to explore the galaxy when we don't know where the Klingons are and NAV to reach the Klingons when we know its location. To explore the galaxy, we calculate the best quadrant to travel to by three factors. Each factor is represented by each gene element. All three factors are: the radius of how far we would like to move (g_{16}), the preference to explore the galaxy nearby or far away (g_{17}, and how many unknown quadrants can be uncovered with the move (g_{18}). Once we know the location(s) of the Klingon(s), we examine all their discovered locations. To find the best location to travel to, the last two factors also apply to the NAV command, except for the radius factor. Both factors are represented by g_{19} and g_{20}.

5.2.5.2 Game Score Evaluation

Previously, we suggest a general guideline to design the game scores with Eq.(5.2) in section 5.2.2. Here, we show our implementation of the game scores for Star Trek in Eq.(5.4).

$$\begin{aligned} game\ score = & \ 0.60 * \#destroyed\_enemies \\ & +0.30 * \#found\_enemies \\ & +0.10 * (win\_game\ ?\ remaining\_time : 0.0) \end{aligned} \tag{5.4}$$

When the game is over, the game scores for fitness evaluation are calculated as the weighted sum of three normalized game statistics. We clarify all three statistics terms denoting to the game scores as below:

1. The number of destroyed Klingons is selected to motivate destroying more Klingons, which eventually leads to winning a game. Due to its highest importance, we assign the largest weight to this term.

2. The number of Klingons discovered in the galaxy is chosen to encourage better exploration. We cannot win a game unless we find all Klingons. This is the second most important factor to win the game.

3. The remaining time after we destroy all Klingons is also a helpful measure. It is to distinguish the best winner from average winners. If two player models win the same game, we prefer the one spending less time. When a game become more difficult with more Klingons, the extra remaining time could be used to search for and destroy the added Klingons. This measure is less significant than the above two because it is not directly related to winning the game. Therefore, we assign the smallest weight to this term. Unlike the above two terms where there are no conditional statements, we apply this term only when the model wins the game. This bonus point is also an incentive to win the game.

All three statistical terms above must be normalized first before applying game score equation. This is to assure that the game scores are rational over varying game parameters.

5.3 Experiments on Evolving Player Model

We setup this experiment to verify that we can evolve our player model parameters with an EC optimization process to improve the performance of the player model automatically. We also compare the improvement made by the EC optimization with the one made by an expert's tuning, in the same player model parameters.

According to the Star Trek gameplay described in section A.2, we select two game parameters to control game difficulty in our experiments.

1. **The game time limit** (for destroying all Klingons). The game time limit has a strong impact on the game difficulty. Giving more game time to a player means the game becomes easier to win, and vice versa, regardless of how the game map is initialized. The relationship between game time and game difficulty is likely to be in a reversely linear proportion.

2. **The number of Klingons**. The game difficulty also depends on the number of Klingons and their distribution throughout the galaxy. In general, giving more Klingon spaceships means the game becomes harder to win, and vice versa. However, the relationship between the number of Klingons and game difficulty is not in a linear fashion. This is due to a fixed size of the galaxy space. A fewer number of Klingons may make the game harder in surveying the galaxy to discover all Klingons. This is because the ratio of the occupied quadrants is very low. A large number of Klingons, however, may make the game harder in destroying all of them. In fact, the distribution of game objects in a galaxy map play an important role in game difficulty. To create a game map, the game environment places all game objects, comprising Starship Enterprise, Klingons, Starbases and Stars, randomly. We control the seed of pseudo-random number generator with the unique game ID to make sure that each game contains a unique galaxy map. Given the same game ID, we design the map initialization in a way that the previously existing game objects are placed at the same random locations, while the recently added game objects are placed at arbitrary new locations. Therefore, the relationship between the number of Klingons and game difficulty is quite in proportion within the game with the same ID. The relationship does not apply across the games with different ID, however.

Please note that, in this experiment, the number of Starbases has minimal effect on the game difficulty because it is not directly related to the game objective, i.e. destroying all opponents within a limited time. Hence, we fix the number of Starbases at three units similar to the default value in the original Star Trek game then adjust only game time and the number of Klingons in our experiments.

5.3.1 Experimental Setups

In our experiments, player model parameters are both automatically optimized by EC optimization and manually adjusted by a skilled game player. We then compare the fitness values from both methods to see if the EC optimization yields a better model performance than the manual tuning.

5.3.1.1 Game Difficulty Levels

We simulated our player model at various levels of game difficulty. Each game difficulty is obtained by altering two game parameters: *the game time* and *the number of Klingons*. Decreasing the game time or increasing the number of Klingons result to more difficult games. The number of Starbases was fixed at three due to its minimal impact on game difficulty.

We characterize game difficulty into three levels: *easy*, *normal*, and *hard*. For each game parameter, we choose the three values at an equal interval that is large enough to distinguish the three levels of game difficulty.

A unit of game time in Star Trek game is called Stardate. One Stardate is approximately equal to a travel across one quadrant. Because the Starship Enterprise can scan the current and its surrounding eight quadrants to see the number of game objects located inside, it takes at least 20 Stardates to survey the entire galaxy when the Starship Enterprise is in a perfect condition, i.e. with no damaged devices. Thus, we assume that 40 Stardates is a moderate time to complete the game mission, so the settings of 30, 40 and 50 are used for hard, normal, and easy level, respectively.

We look at the suitable number of Klingon spaceships in terms of its distribution within an 8×8 galaxy space. The Klingons are distributed throughout 64 quadrants randomly with a maximum of three in one quadrant. To give a suitable occupation rate, we use settings of 10, 15, and 20 Klingon spaceships for easy, normal, and hard level, respectively.

The combination of game parameters settings establishes the total nine pairs of game parameters at several levels across the whole range of game difficulty. All pairs of game parameter settings are presented in Table 5.2.

Table 5.2: Nine pairs of game parameter settings at various game difficulty to evaluate a player model. Two game parameters are controlled to create different game difficulty levels: the game time (t) and the number of Klingons (k). We assign three difficulty levels to each value of game parameters: easy, normal (norm), and hard level.

game parameters	t = 30 Stardates	t = 40 Stardates	t = 50 Stardates
k = 10 Klingons	k=easy, t=hard	k=easy, t=norm	k=easy, t=easy
k = 15 Klingons	k=norm, t=hard	k=norm, t=norm	k=norm, t=easy
k = 20 Klingons	k=hard, t=hard	k=hard, t=norm	k=hard, t=easy

5.3.1.2 Optimization Settings

For a DE optimization algorithm, 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 21 player model parameters. The population size is approximately twice the size of optimized parameters, which is normally recommended as the conventional practice. Each individual in the population is represented by an array of 21 real numbers, as illustrated in Fig. 5.4.

Table 5.3: Key experiment setups.

player model parameter size	**population size**	**DE settings**	**DE runs**
21	40	DE/best/1, binomial crossover, F=0.9, CR=0.8	50

For a given number of Klingons, we initialized ten galaxy maps with different Klingons' distributions. The map distribution is incremental, i.e. a new Klingon is placed in a random location while the existing Klingons' locations remain 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 unique initial galaxy maps. At the end of each game, the game scores are computed. At the end of all ten games, all game scores are averaged and the winning ratio among the ten games is calculated. We use the summation of both terms as a fitness evaluation for each DE individual.

5.3.1.3 Simulation Setups

We conducted 18 sets of simulations to evaluate the tuning capabilities of our player model: nine for DE-optimized tunings and the other nine, from the same game parameter settings, 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. We run each set three times for a specific number of Klingons with 30, 40, and 50 Stardates. All simulations play the same set of ten unique galaxy maps, then compared the results with the corresponding DE-optimized tunings.

5.3.2 Experimental Results

In this subsection, we show the performance comparison between the manual tunings and DE-optimized tunings. We made the nine comparisons showing in Table 5.2. As the EC algorithm usually outperforms a human's performance after its evolving population reach the stable state, we examine both initial evaluation and stable-state evaluation of DE-optimized player models.

The 3-way comparison among manual tunings, initial-state, and stable-state of DE-optimized tunings is best examined in a table form, presented in Table 5.4 and 5.5, showing the results in symbols and numerical values. Each row shows the comparison for each difficulty level. As our data do not form in a normal distribution, we applied the Friedman test and the Holm's multiple comparison test over the 50-runs data to validate the consistency of the results.

The evolution of DE-optimized tunings from the initial state to the stable state, at the 200th, is best observed visually as a fitness curve. In all resulting graphs, we plotted nine fitness curves from DE-optimized tunings on the right-hand side of the graph; one for each difficulty levels. For side-by-side visual comparison, we also plotted nine fitness points from manual tunings on the left-hand side.

We compared the performance of both tunings using (1) *the best fitness scores* and (2) *the maximum numbers of winning games* evaluated from the tuning models. The results from our experiments are as follow.

5.3.2.1 Best Fitness Score

The best fitness scores of manual tuning represents the fitness value evaluated from the player model tuned by the authors. For a single run of DE optimization, the best fitness scores represent the highest value of fitness evaluation among all individuals in DE population. With 50 DE runs in our experiments, we use the average value of the best scores in each run.

Table 5.4: 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**				
k=10, t=50	1.5338_{1st}	$\ll$	1.6741_{man}	$\ll$	1.9343_{200th}
k=10, t=40	1.2356_{1st}	$\ll$	1.4446_{man}	$\ll$	1.9108_{200th}
k=15, t=50	1.1759_{1st}	$\ll$	1.6276_{man}	$\ll$	1.7724_{200th}
k=20, t=50	0.7987_{man}	$\approx$	0.8173_{1st}	$\ll$	1.3838_{200th}
k=15, t=40	0.8070_{1st}	$\ll$	1.0430_{man}	$\ll$	1.2195_{200th}
k=10, t=30	0.8473_{1st}	$\approx$	0.8511_{man}	$\ll$	1.1640_{200th}
k=20, t=40	0.5950_{1st}	$<$	0.6000_{man}	$\ll$	0.7960_{200th}
k=15, t=30	0.5560_{1st}	$<$	0.5660_{man}	$\ll$	0.6931_{200th}
k=20, t=30	0.4740_{man}	$\ll$	0.4933_{1st}	$\ll$	0.5808_{200th}

According to our simulations at nine levels of various game difficulty, Fig. 5.5 shows the best fitness scores for both manual tunings (left points) and DE-optimized tunings (right curves). Table 5.4 displays 3-way statistical comparison among manual tunings, DE-optimized tunings at the 1st and 200th generations.

5.3.2.2 The Maximum Number of Wins

The main goal for player model optimization is to increase more wins for our model. However, we cannot simply use the number of wins directly for a fitness evaluation because its discrete value lacks the finely guiding control toward the main goal. Therefore, the continuous fitness evaluation in Eq.(5.1) is suggested. Even though the best fitness scores seem to represent the best player model nicely, there are times when the value misinforms us about the number of wins. According to the game score calculation in Eq.(5.4), a frequent of close defeats may overtake a tiny

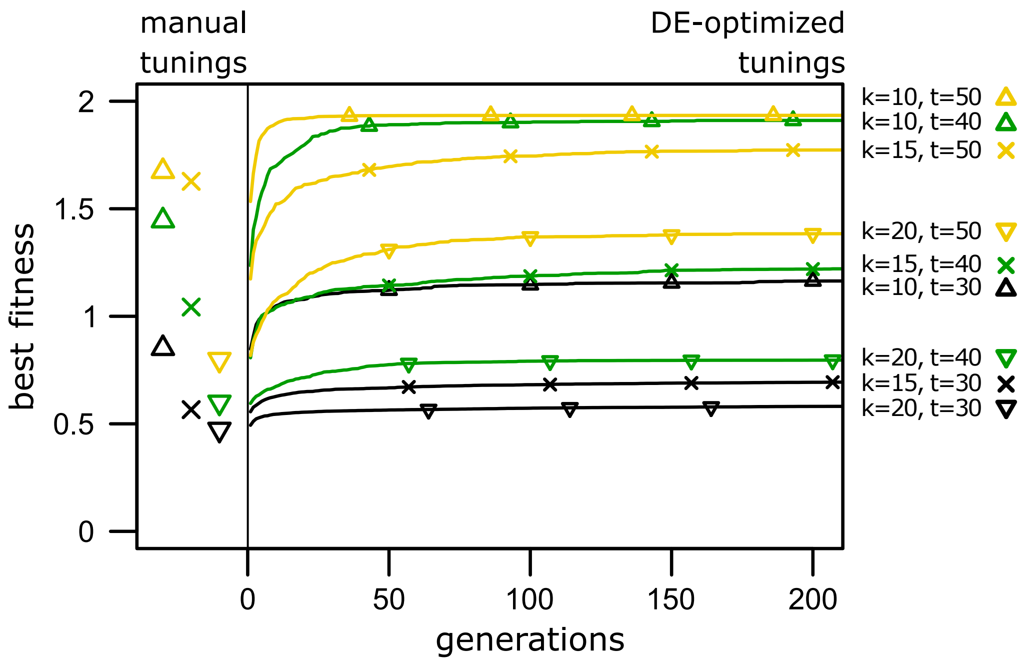


Figure 5.5: Experiment results of best fitness score from evolving player model.

close win. Hence, a maximum number of wins is another measure for player model performance. Furthermore, the fitness scores should be coherent with the number of wins to exhibit its optimization efficiency.

Similar to the computation of the best fitness scores, the maximum number of wins for DE optimization is the average from 50 DE runs.

According to our simulations at nine levels of various game difficulty, Fig. 5.6 shows the maximum number of the games won, for both manual tunings (left points) and DE-optimized tunings (right curves). Table 5.5 displays 3-way statistical comparison among manual tunings, DE-optimized tunings at the 1$^{\text{st}}$ and 200$^{\text{th}}$ generations.

5.3.3 Discussions

In terms of both fitness scores and a number of wins, all 50 runs of DE-optimized player model performed better than the manual tunings in almost all difficulty levels. The only exception is in the most difficult game level, with the maximum number of Klingons (20) and the minimum game time (30 Stardates); here the best fitness improves very little and cannot win any single game.

Table 5.4 and Table 5.5 compares the simulation results among the three groups of parameter tunings: manual tunings, initial generation, and stable generation of DE-optimized tunings. In the beginning, the manual tuning made by an expert

Table 5.5: 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.

game parameters	**Number of won games (out of 10)**				
k=10, t=50	7.12_{1st}	$\ll$	8_{man}	$\ll$	10.00_{200th}
k=10, t=40	4.66_{1st}	$\ll$	6_{man}	$\ll$	9.94_{200th}
k=15, t=50	4.42_{1st}	$\ll$	8_{man}	$\ll$	9.08_{200th}
k=20, t=50	1_{man}	$\ll$	1.58_{1st}	$\ll$	5.88_{200th}
k=15, t=40	1.62_{1st}	$\ll$	3_{man}	$\ll$	4.62_{200th}
k=10, t=30	1_{man}	$\ll$	1.56_{1st}	$\ll$	3.84_{200th}
k=20, t=40	0_{man}	$\approx$	0.04_{1st}	$\ll$	0.9_{200th}
k=15, t=30	0_{man}	$\approx$	0_{1st}	$\ll$	0.1_{200th}
k=20, t=30	0_{man}	N/A	0_{1st}	N/A	0_{200th}

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 in 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 the ones achieved via human adjustment. Even though human experts not only play the game excellently but also design the FS rules reasonably well, the interpretation of FS inputs with membership functions may not be an easy task.

From Table 5.4 and Table 5.5, 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, having the most optimization improvement. The last three rows are hard games with a less than 10% winning rate, showing no improvement at all. This is, most likely, because the hard games have such a low winning rate at the beginning, resulting to a weak selection pressure to search for better solutions. On the contrary, the easy games have higher initial winning rates, so they generate a stronger selection pressure to search for optimal solutions.

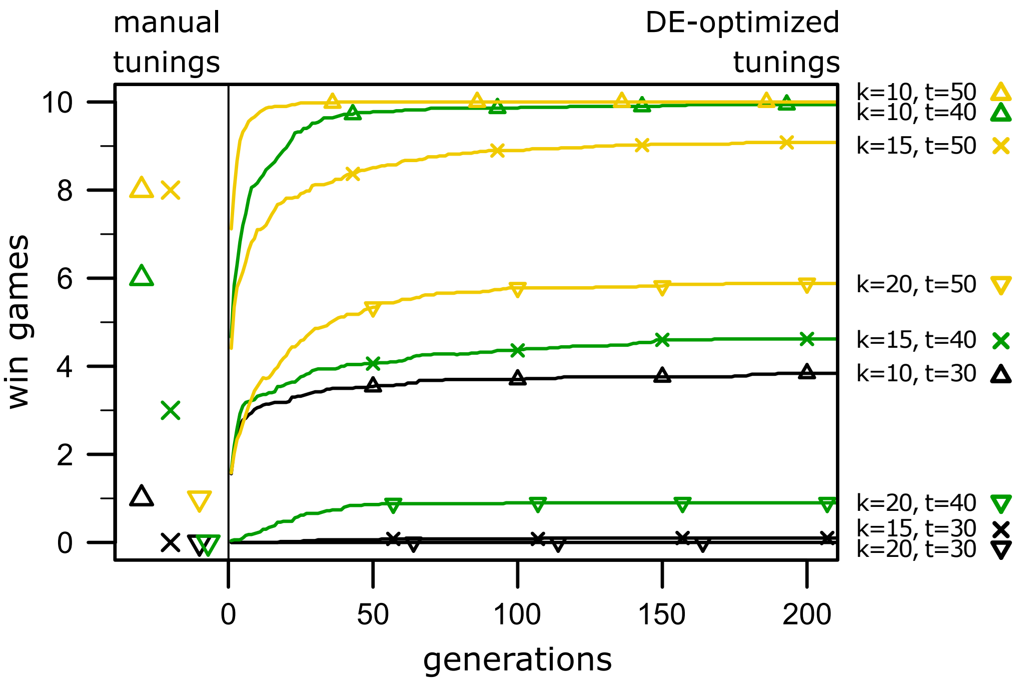


Figure 5.6: Experiment results of win games from evolving player model.

5.4 Chapter Summary

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. The optimized player model generated better fitness scores than conventional tunings by an expert, 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 an FS decision-making system and an EC-based approach to parameter optimization, which is not just limited to the DE technique, is a feasible framework with semi-automatic collaborations toward the automatic game parameter tuning.

The concepts and experiments explained in this chapter were presented at the 12th International Conference on Innovative Computing, Information and Control on August 29, 2017. The article of the same content was published in International Journal of Innovative Computing, Information and Control in December 2017 [60].

Chapter 6

Learning Player Model by Gradually Increasing Game Difficulty Levels

6.1 Introduction

The previous chapter explains how our DE-optimized player model evolves over various settings of game parameters. At the stable state of DE optimization, all evolving player models improve over their initial performance in every static setting of game parameters. In this chapter, we examine the results of DE-evolving player model on dynamic game parameter settings.

With manual game parameter adjustment, we can freely control game difficulty. When altering game difficulty systematically, we investigate performance responses of the game player model while it evolves. The positive responses with controllable game difficulty are a valuable step toward automatic game parameter tuning.

The main objective of this chapter is to study the relationship between manual game parameter adjustment and evolution of the game player model. We also explore the essential factors for the success of human-guided game parameter adjustment. This will give us a better understanding about the mutual evolution process (see Fig. 3.2), which is the foundation of the automatic game parameter tuning in our framework.

Following this section, we discuss the idea behind the manual game parameter adjustment in section 6.2. This section also includes some issues on the adjustment methods. In section 6.3, we conduct experiments to show several aspects of evolving player model with game difficulty increment. We summarize our findings on manual game parameter adjustment in section 6.4.

6.2 Incremental Learnings of Game Difficulty

6.2.1 Scale-space Filtering

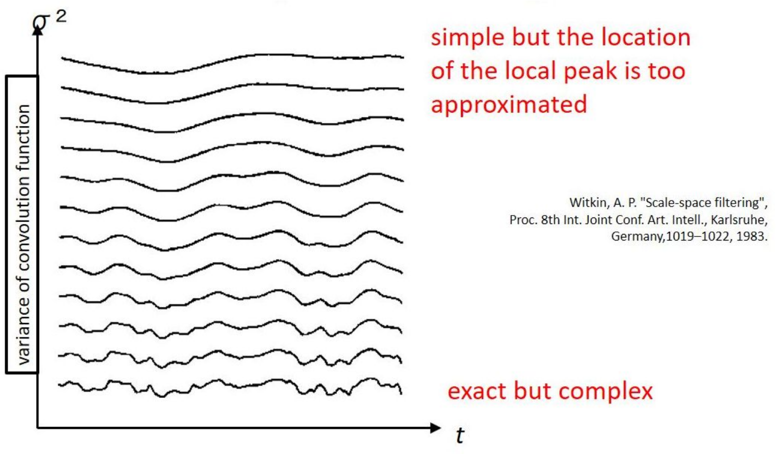
The DE optimization process searches for the optimal values of player model parameters that represent the best player in a particular game. When the optimization evolves the player model for that game, it searches for the parameter landscape that belongs to the game. The process hunts for the optimal evaluation point in the landscape given by the fitness evaluation function. When the game changes, the parameter landscape changes accordingly.

Our assumption for the changing landscape is that when the game slightly changes, its landscape will, just a little, change correspondingly. Fortunately, we can control the game difficulty with its game parameters. That means, for a specific game difficulty, once the optimization process finds the optimal parameter solution, it should take less effort for the process to find a new optimal solution at the slightly-changed game difficulty.

This assumption is inspired by the canonical work of the late Andrew Witkin in 1983, named scale-space filtering [62]. As the first step of his proposed signal-processing technique, Witkin applied Gaussian smoothing repetitively with increasing σ parameters over the same waveform. Fig. 6.2 shows the result of this sequential Gaussian smoothing. Each row represents a signal. The original signal locates at the bottom of the figure. The upper signals are Gaussian-smoothed version of the lower ones. The top row is the smoothest signal. It contains the largest Gaussian convolution (σ value) in the figure.

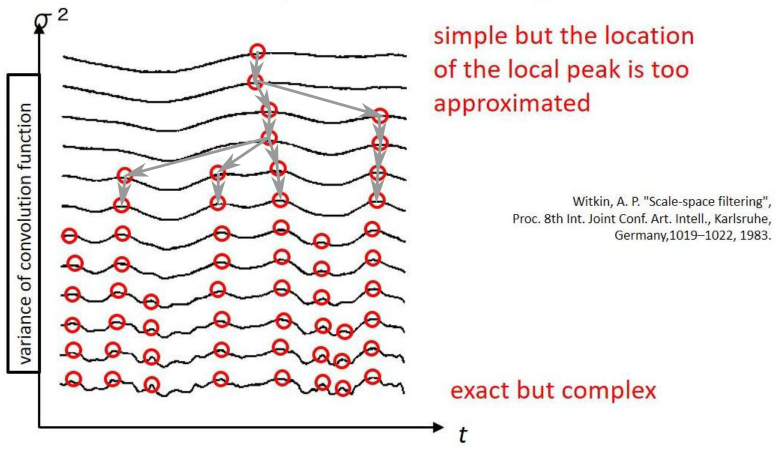
We can view this figure and the smoothing filter in the different way from top to bottom. Assume that the waveform in this figure is a parameter landscape. A smoother waveform at the upper row represents a parameter landscape for an easier game. In such landscape, it is quick and simple to locate the optimal position. Later, in the lower row, the parameter landscape becomes more complicated in a harder game. We can start exploring the complicated landscape from this formerly optimal position. It will be quicker and simpler to find out the new optimal position than to start the landscape exploration from random points.

Using this analogy for our problem, it is simpler to start optimizing the player model in an easy-level game. We then continue optimizing the model with gradually-increasing game difficulty. This approach should take less optimization time or gain more performance improvement at the target difficulty level, when compared to working straight on a hard-level game.



σ^2
variance of convolution function
simple but the location of the local peak is too approximated
Witkin, A. P. "Scale-space filtering", Proc. 8th Int. Joint Conf. Art. Intell., Karlsruhe, Germany,1019–1022, 1983.
exact but complex
t

Figure 6.1: Sequential Gaussian smoothing of a waveform. The original waveform is on the bottom row. The upper waveforms are smoother than the lower ones, due to increasing σ values in Gaussian convolution.



σ^2
variance of convolution function
simple but the location of the local peak is too approximated
Witkin, A. P. "Scale-space filtering", Proc. 8th Int. Joint Conf. Art. Intell., Karlsruhe, Germany,1019–1022, 1983.
exact but complex
t

Figure 6.2: Waveform as a searching parameter landscape. It is easier to find an optimal solution in a smoother searching landscape (upper waveforms). Searching around this optimal position may help to find a new optimal solution easier and faster in a more complex landscape (lower waveforms).

We call this approach *a gradual incremental learning*. It comes from an incremental learning approach in machine learning (ML). The goal of incremental learning ML is to train the learning model to adapt to new data while still remember all its existing knowledge. This is to avoid retraining the model. It is also similar to our gradual incremental learning approach. The exception lies in the condition that we require a small number of differences in each searching landscape.

6.2.2 Issues in Gradual Incremental Learning

Basically, there are two major concerns when applying gradual incremental learning in parameter optimization. Both issues may result to differences in performance improvement on the optimization or the overall time required to optimize the system. These two major concerns are:

- **Sizing Policy**: The sizing policy is related to the *gradual* requirement in the name of the approach. It is about the sizes of incremental changes that are suitable for the learning.

  The suitable sizes in gradual incremental learning vary, in general, depending on the optimizing parameters. On the one hand, the size should be small enough to generate smooth transition in the searching landscape. This helps the optimization process to locate new optimal position easily. On the other hand, the size should be large enough to allow significant changes in the fitness evaluation.

- **Timing Policy**: The timing policy is how we determine the right time to apply the incremental changes. On the one hand, if we apply the changes too soon, the system may not be fully evolved. This may result to poor overall performance at the end of the optimization process. On the other hand, if we apply the changes too late, the system becomes less efficient due to a waste of time.

  The simplest timing policy is a fixed-interval policy. We supply the incremental changes in a specific time period. The rule of thumb is that this time period should provide a balance between a proper system evolution and an efficient amount of time. Another approach for the timing policy is to use some other measures to manage the changing interval. Normally, the fitness evaluation in the optimization process helps to decide the appropriate time to change. There are many implementations in the approach, e.g. making a change when the n-best fitness scores are stable for a specific time period, etc.

In this chapter, we set up experiments to explore the gradual incremental learning with a focus on timing policy. Due to a limited range in Star Trek game parameters, it is not suitable to use the game for the study of sizing policy. Although the system improvement is a main advantage of incremental learning ML, we do not concentrate on the most improved standpoint. Rather, we are interested in a stable improvement, even at the minimum degree, that provides valuable feedbacks for the benefit of mutual evolution process. This is our key mechanism for automatic parameter tuning framework.

6.3 Experiments on Incremental Learnings of Game Difficulty

Our experiments show performance responses of the player model when game difficulty is gradually increased. We adjust the game parameters to control the game difficulty from easier levels to harder ones while a DE optimization evolves the player model. We conduct two related experiments in this chapter examining the timing policy for gradual incremental learning.

For the experiments, we first try to explore the fixed-interval timing policy to adjust the game difficulty in subsection 6.3.2. We then experiment on an adaptive-interval timing policy in subsection 6.3.3. Both policies are compared and discussed at the end of the section.

6.3.1 Difficulty Levels and Star Trek Game Parameters

In this chapter, we still use Star Trek game, described in Appendix A, as a test bed in the experiments. We alter the game time to create levels of game difficulty. Because a player must destroy all Klingons within a given game time, it is easier to win Star Trek game when longer game time is provided, and vice versa. Therefore, the game time establishes a direct relationship to the game difficulty. Steadily decreasing Star Trek game time is proportional to gradually increasing game difficulty.

Unlike the game time in Star Trek, the number of Klingons has a non-linear relationship with the game difficulty. It is both the number of Klingons and the distribution pattern of this number in the galaxy that jointly manipulate the game difficulty. Thus, we set the number of Klingon spaceships to 15 and the number of Starbases to 3, constantly. These values are a regular setting in normal-level games, according to the previous experiments in section 5.3.

Table 6.1: Classification of three-levels game difficulty.

game difficulty	**players' winning percentage**
easy level	more than 75%
normal level	25% - 75%
hard level	less than 25%

Generally, we define the game difficulty level from the game players' *winning percentage*, a percentage of games a player has won. For simplicity, we classify Star Trek into three levels of difficulty: easy-level, normal-level, and hard-level games. An easy-level game is the game that more than 75 percent of the players can win. A hard-level game is the game that less than 25 percent of players can win. A normal-level game is the game that most of the averaged players, i.e. between 25 to 75 percent, can win. Table 6.1 shows the classification of three-level game difficulty.

Table 6.2: Three levels of game difficulty increment.

game difficulty increment	**target game parameters**			**initial**	**decay**	**manually**
	Klingons	**Starbase**	**game time**	**time**	**rate**	**adjusted time**
toward easy level	15 (static)	3 (static)	50	60	-2	60,58,56,54,52,50
toward normal level	15 (static)	3 (static)	40	50	-2	50,48,46,44,42,40
toward hard level	15 (static)	3 (static)	30	40	-2	40,38,36,34,32,30

For our incremental learning, we alter the game time parameter toward a target level of game difficulty. All three target levels in the experiment is shown in Table 6.2. For each difficulty level, we start playing a game at one level easier than the target level. We then increase the game difficulty by decreasing game time manually for the evolving player model. Because the game time is a discrete unit, we decrease it at a constant decay rate. That also results to the uniform increment of game difficulty. We select the decay rate at two Stardates, a game time unit in Star Trek game, for each game time reduction. Two Stardates are roughly equal to two playing turns. This amount is short enough for a gradual increment. It is also not too short to make no differences in the final outcome of the game. Table 6.2 summaries game parameter adjustment in our experiments.

For all experiments in this chapter, we follow this game parameter adjustment scheme as a gradual increment of game difficulty. However, when to alter the game parameters depends on the timing policy. The following experiments in subsection 6.3.2 and 6.3.3 investigate a fixed-interval timing policy and an adaptive-interval timing policy, respectively.

6.3.2 Fixed-interval Incremental Learning of Game Difficulty

We explore the timing policy for our incremental learning in this experiment. We evolve the game player models while increasing the game difficulty gradually, at a fixed interval of DE generation. We conduct the experiments for three different game difficulty levels, i.e. easy-level, normal-level, and hard-level games. We investigate the influence of game difficulty levels upon the performance of the player models.

6.3.2.1 Experimental Setups

For gradual incremental learning, we adjust game parameters according to Table. 6.2 every 100 DE generations. According to Fig. 5.5, at the 100th DE generations, the evolving player model already reaches their stable state in all three game times with 15 Klingons. Therefore, the adjusting time at every 100 generations is long enough to evolve the model.

Table 6.3: Key experiment setups.

player model parameter size	**population size**	**DE settings**	**DE runs**
21	40	DE/best/1, binomial crossover, F=0.9, CR=0.8	30

We used the same game player model and its setups from the previous chapter (see subsection 5.3.1). Table 6.3 shows the key parameters for the setups. We applied the gradual incremental learning explained earlier at every 100 DE generations. We conducted 30 DE runs in the experiments for reliable statistical measures.

6.3.2.2 Experimental Results

Figure 6.3 illustrates the evolution of 30-run averaged best fitness scores from six game player models. Three player models with gradual incremental learning (dashed lines) produce a saw-toothed shape with five value-drops in every 100 DE generations. This corresponds to five incremental changes in our experimental setup: a reduction of two Stardates from a range of ten Stardates at a fixed interval.

Table 6.4 shows the best fitness scores at the 700th DE generation, which is a stable state in all experiments. The player model with gradual incremental learning works poorly with the hard-level games. Its best fitness scores are lower than the standard player model at 5% significant difference. For easy-level and normal-level games, although the gradual incremental learning shows higher best fitness scores, its p-value shows no significant difference. Therefore, we cannot conclude the benefit on gradual incremental learning at the fixed interval of 100 DE generations.

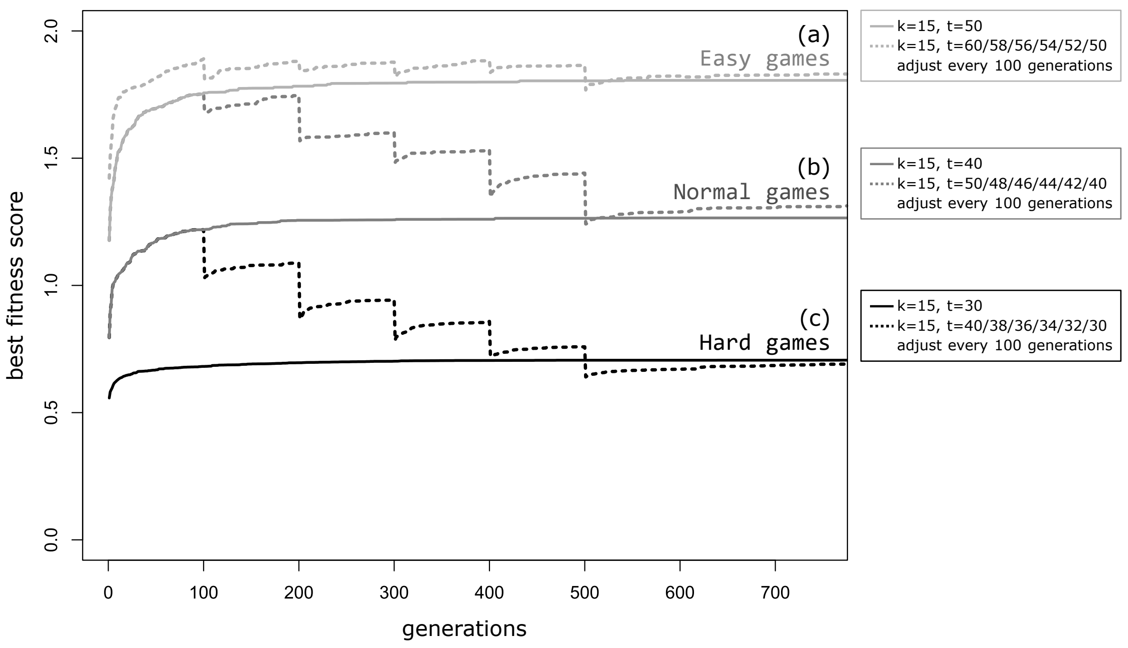


Figure 6.3: Three comparisons between a standard DE-optimized player model (solid line) and a DE-optimized player model with gradual incremental learning of game difficulty (dashed line), at (a) easy, (b) normal, and (c) hard level of game difficulty, respectively. With fixed-interval timing policy, the game difficulty increases by reducing two Stardates every 100 DE generations.

Table 6.4: Best fitness scores between a standard DE-optimized player model and a DE-optimized player model with gradual incremental learning from three levels of game difficulty, at the 700^{th} DE generation. The p-value in the last column is calculated from Wilcoxon signed rank test.

	30-runs averaged best fitness score		
game difficulty level	**standard**	**gradual incremental learning**	**p-value**
easy-level games	1.806	*1.827*	0.092
normal-level games	1.265	*1.306*	0.129
hard-level games	*0.706*	0.685	0.043

6.3.2.3 Discussion

The experimental results show the significant drawbacks of gradual incremental learning approach in the hard-level games, as shown in the bottom of Fig. 6.3. This is probably due to its very low winning percentage which creates a weak selection pressure to improve the model. The games are so hard that the player model can hardly win. This results to the lack of guiding direction toward the optimal location in the search space. Starting this hard-level gradual learning from the normal-level games does not help improve the winning percentage. Instead, for our assumption, the learning may gather DE individuals to a wrong area of solutions, which are practical for evolving at normal level yet impractical for learning toward the hard level. Without gradual incremental learning, the individuals are initially distributed randomly throughout the solution space. This may have a higher chance to find a solution for hard-level games rather than having individuals gather around a specific area.

Looking at the results in terms of time consuming, it is noticeable that a fixed interval of 100 DE generations takes much longer time to reach its stable state, when compared to a standard player model. We need to examine a different approach for timing policy that saves more time. We try an adaptive-interval timing policy in the following experiment.

6.3.3 Adaptive-interval Incremental Learning of Game Difficulty

We continue exploring a new timing policy for gradual incremental learning in this experiment. The inflexibility in fixed-interval timing policy, i.e. the trial-and-error effort to set the practical interval value, makes the approach ineffective due to an excessive waste of time. Instead, we apply adaptive-interval timing policy that helps to save more time.

One good practice in timing policy is to adjust the system when it reaches a stable state. There are many indicators to keep us informed about this. A simple way is to use the fitness evaluation scores. When the fitness scores are unchanged for a period of time, we may assume that the system is somewhat in its stable state.

There are two major elements to control this *stable-fitness* timing policy. One is the specific time period, in DE generation, when the fitness scores keep constant. The other element is the type of fitness scores used in the measure of a stable state, e.g. only the best fitness scores or the average of the best n scores in the population. In this experiment, we concentrate on the control of stable time only. We make a change when the best fitness scores are stable for g DE generations.

6.3.3.1 Experimental Setups

The experimental setups for adaptive-interval timing policy are similar to the previous experiments in fixed-interval (see subsection 6.3.2.1). However, we add a time period control to check whether the player model is now stable and ready for the changes. In the experiments, we observe the best fitness scores to be constant for 5, 10, and 25 consecutive DE generations before increasing the game difficulty.

At the moment, we focus our experiments on incremental learning toward the normal-level game difficulty. This is because a wider range of game players enjoy the normal-level settings than the other settings. Therefore, in this experiment, our target game parameter setting is the Star Trek game with 15 Klingon spaceships, 3 Starbases and 40 Stardates game time.

6.3.3.2 Experimental Results

Figure 6.4 illustrates the evolution of the best fitness scores in three DE-optimized player models with adaptive-interval incremental learning. We apply the stable-fitness timing policy to all three player models. Three stable intervals of 5, 10, and 25 DE generations are shown via a dotted line, a thin line, and a dashed line, respectively.

The value of best fitness scores in the figure is an average of best individuals in 30 DE runs. In each run, the game difficulty changes at different time. Furthermore, in each time, the performance first drops sharply at the changing point then gradually gains its value back to a certain strength, as the system adapts to the harder games. An individual DE run with adaptive-interval policy has its evolution path in a saw-toothed shape, similar to the model with fixed-interval policy (see Fig. 6.3), except that the saw-tooth interval is unevenly distributed.

The accumulated graph of all 30 runs, which is similar to their average, is illustrated in Fig. 6.4. The shape of each set of accumulated fitness scores all look alike. We can divide this shape into three major stages: a steep rise, a decline, and a recovery. In the first stage, the fitness scores increase rapidly due to the easier level of game difficulty. All 30 runs are in this early stage at the same time. Hence, we notice a very steep graph in all three player models. At the peak of this stage, some DE runs reach their first stable state and their fitness scores drop due to game difficulty increment. It is where the uneven saw tooth in the individual run begins to take effect. In this second stage, the graph falls at different rate because each DE run encounters harder games at different time. A short interval of a stable state (stable=5) creates a sharp fall while a long interval (stable=25) produces a steady decline. At the bottom of this stage, all DE runs reach the target game difficulty. From this point onward, the graph shows the recovery of the accumulated fitness scores. The speed of recovery highly depends on the stable interval as well. A short fall in the second stage creates a large recovery in the third stage because all DE runs fall nearly at the same time. Thus, they will recover at the same time as well.

On the contrary, a steady decline in the second stage creates a slow recovery because all runs gradually recover over a longer period of time. All three player models share a similar shape of the best fitness graph. However, the shape has different rates of decline and recovery periods. It is in the recovery stage that the system shows its performance improvement.

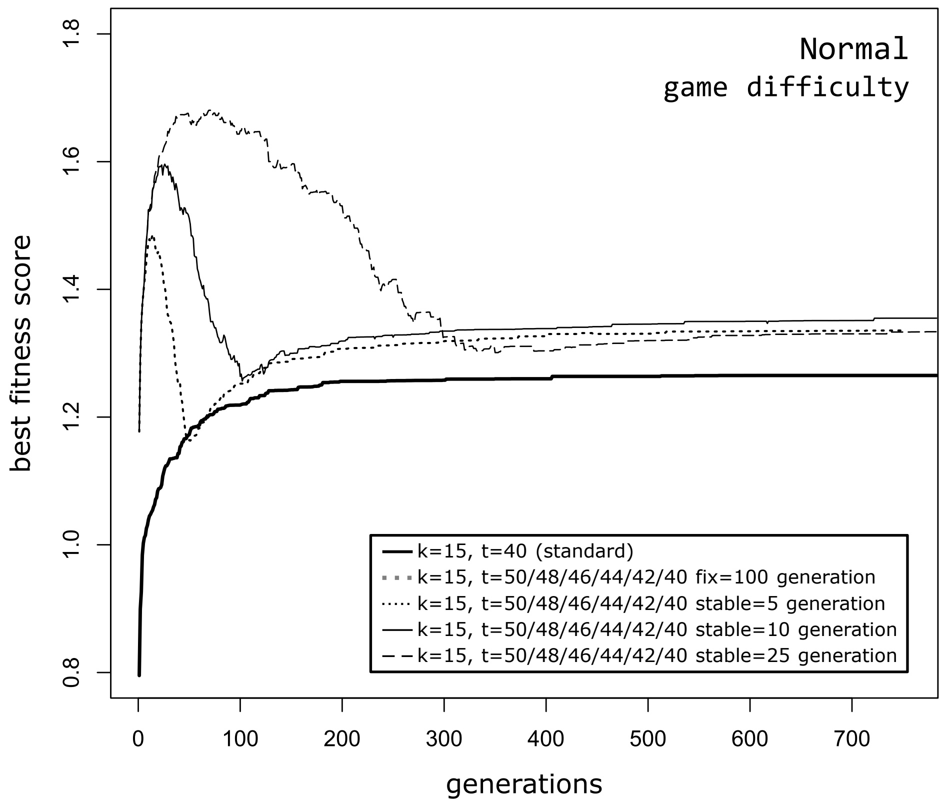


Figure 6.4: Best fitness score comparison between a simple DE-optimized player model and three DE-optimized player models with adaptive gradual incremental learning of game difficulty. The game difficulty increases by reducing two Stardate adaptively when the best fitness scores are stable continuously for 5 (dotted line), 10 (thin line), and 25 (dashed line) DE generations. The game time reduces from 50 Stardates in easy-level games down to 40 Stardates in normal-level games.

6.3.3.3 Discussion

With the stable interval of 5 DE generations, the incremental changes occur very quickly. It creates small and narrow overshoot. It looks like the system is not fully optimized yet. On a contrary, when compared to the stable interval of 25 DE generations, the changes occur in a longer time period. Its overshoot is larger in both height and width. However, it reaches the recovery stage very late. It is ineffective in terms of time consumption. The suitable interval, among these three cases, is the stable interval of 10 DE generations. It shows a good shape compared to the other two intervals. Its shape also matches with the base shape of the player model without incremental learning. Its decline stage ends nearly at the beginning of steady state of the standard player model. It has a nice recovery time period that improves its performance over the standard model.

Figure 6.5 and Table. 6.5 show a comparison between player models with fixed-interval and adaptive-interval timing policy for normal-level game difficulty. All player models with adaptive-interval policy show higher best fitness scores than the standard player model with significant difference at 1% level.

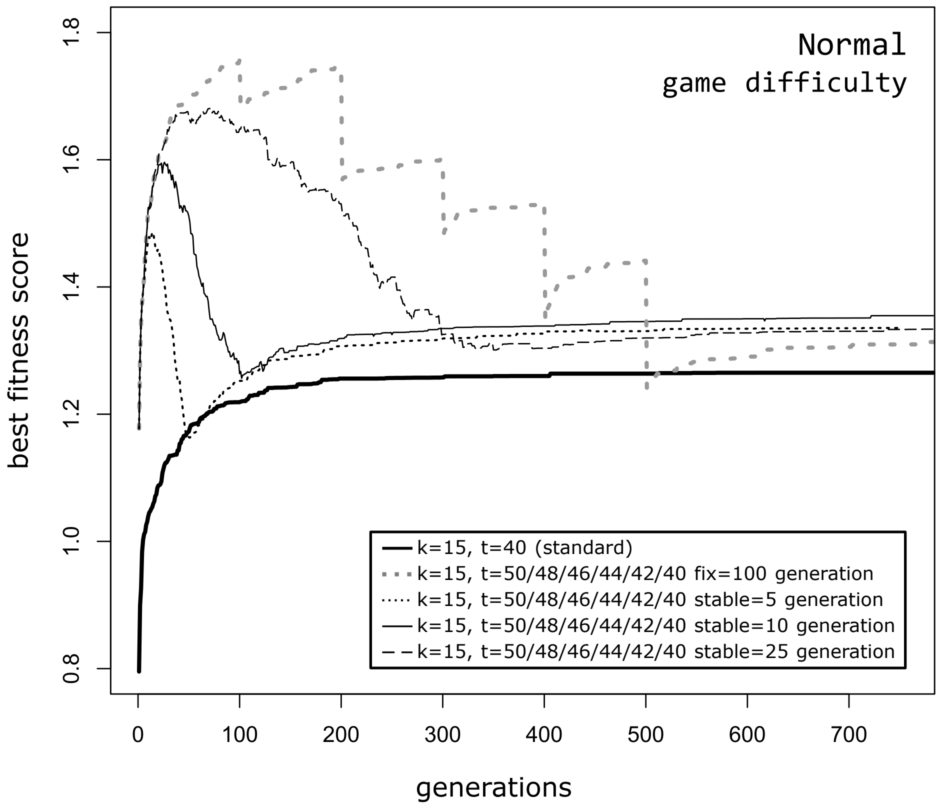


Figure 6.5: Comparison between a fixed-interval (dotted line) and an adaptive-interval (solid thin line) timing policy for gradual incremental learning.

Table 6.5: Best fitness scores from fixed-interval and adaptive-interval timing policies in normal level of game difficulty, at the 700th DE generation. The p-value in the last column is calculated from Wilcoxon signed rank test.

timing policy	**best fitness score**	**p-value**
- (standard)	1.265	
change every 100 generations (fixed)	1.306	0.129
change when 5 generations are stable (adaptive)	1.335	0.001
change when 10 generations are stable (adaptive)	*1.351*	*0.000*
change when 25 generations are stable (adaptive)	1.330	0.002

It is quite obvious that the adaptive-interval incremental learning can improve the performance of a DE-optimized player model. The improvement is robust. It is a positive indication for a mutual evolution process in our automatic parameter tuning framework.

6.4 Chapter Summary

The manual adjustment on key game parameters allows us to control game difficulty deliberately. With a concept of gradual incremental learning, we can improve the performance of evolving player model by gradually increasing game difficulty. With gradual incremental learning, the player model adapts the knowledge learnt from playing with easier games to play with the harder ones. Using the mentioned knowledge, the model should play harder games more efficiently than a standard player model would. Although gradual incremental learning takes time, selecting a suitable timing policy is the key to its success.

Chapter 7

Discussions

In this section, we discuss some findings along the way on the experiments. Some topics may be add-ons to the main theme, while some are on-going experiments.

7.1 Toward the Generalization of Automatic Game Parameter Tuning

As Star Trek game is used as our experiment test bed, most of the implementations for our methodology focus mainly on the Star Trek game. Considering the game-specific purpose of the methodology, this is not unusual in the game parameter tuning process. However, extending our methodology to cover most of video games in the TBS game genre rather than Star Trek game is not complicated.

Star Trek game represents a typical TBS game which shares the same game design elements discussed in section 3.2.3. The design of Star Trek game covers all four elements of TBS game design. Table 3.1 shows the list of Star Trek game parameters in relation to each design elements for TBS games. In terms of the game and game parameters, adapting our approach used with Star Trek game to other TBS games is quite straightforward with the knowledge of both Star Trek game and another TBS game in target.

For the input interface of a player model, the technique to retrieve game information from the game output varies. Star Trek game is a text-based game and parsing text data to retrieve game information is a reasonable method. With other kinds of game output, e.g. graphic-based, audio-based, haptic-based, etc., an appropriate technique is needed.

Similarly, for the output interface of a player model, the technique to send out game commands also varies depending on the game input. An appropriate output method must be used to match different kinds of game input, e.g. text commands, simulated keypresses, audio outputs, simulated physical movements, etc.

Table 7.1: Summary of our methodology for automatic game tuning (4) compared to three other related researches (1-3) discussed in section 2.2.5.

	Game Title & (Game Genre)	**Gameplay & Game Input**	**Optimized Game Parameters**	**Main Research Objectives**	**Player Model's Simulated Control**	**Key Algorithms**
1	**Flappy Bird** [23] (single-player, minimal action game)	**gameplay:** Control the bird to navigate through a series of pipes, as far as possible. **input:** one-button key press	12 parameters: pipe separation, pipe gap, pipe width, pipe gap location range, gravitational constant, jump velocity, bird speed, world height, bird width and height, change in pipe gap, change in bird speed.	Exploring Game Space and Analysis of Game Difficulty	Randomized simulation of human motor skills on key pressing	Generate game space by survival analysis of score histogram. With the game space, search for a target game difficulty using DE, search for the unique games using GA, etc. **game measure:** distance scores
2	**Spacewar** (a simple clone) [31] (two-player, space-battle action game)	**gameplay:** Control the spaceship and fire a missile to destroy the opponent. **input:** 4 action inputs: Rotate-Clockwise, RotateAnticlockwise, Thrust, Shoot	5 parameters: Maximal ship speed, Thrust speed, Maximal missile speed, Cooldown time, Missile cost, Ship radius	Automatic Playtesting	Monte Carlo tree search (MCTS) based, autonomous game agents from GVG-AI competitions. (same as 3)	Random Mutation Hill-Climber & Multi-Armed Bandit Random Mutation Hill-Climber **game measure:** destroyed opponents & launched missile
3	**Legend of Zelda** (a simple clone) [19] (single-player, action-adventure game)	**gameplay:** Navigate from starting point to the stair while collecting coins & pickaxes and avoiding the attack from 4 tanks. **input:** 4 arrow keys to move, a space bar to throw pickaxes.	8 parameters: Tank Speed, *Score Pickaxe*, *Score Wall Kill*, Pickaxe Value, *Time Bonus*, *Score Gold*, Pickax Limit, Pickax Cooldown	Strategic Diversity	Monte Carlo tree search (MCTS) based, autonomous game agents from GVG-AI competitions. (same as 2)	Random Mutation Hill-Climber **game measure:** optimized game score (collected coins & Pickaxe, destroyed tank). (see Optimized Game Parameters column)
4	**Star Trek** [60] (human-vs-computer, turn-based, strategy game)	**gameplay:** Destory all opponent spaceships within a given mission time. **input:** text commands & command arguments	2 parameters: number of opponents, mission time.	Game Paramter Tuning	FS rule-based decision system	DE to optimize a player model and *(CEA to coevolve between the multi-skilled player model and game)*$^{*}_{in-progress}$. **game measure:** destroyed opponents, found opponents, and remaining win time & winning ratio

However, the EC optimization process is the greatest concern for the generalization of our approach. The success of EC-optimized game player model highly depends on the design of the fitness evaluation and encoding gene for the EC algorithm. These issues are game-specific and its success varies on the game itself as well as the experiences of game developers in the design of FS rules and the implementation of supplementary decision callback functions.

We suggest several techniques to enhance the design of FS rules for a player model in section 4.4, i.e. modular FS tables, FS table re-evaluation, and multi-output FS decision. We also present different ways to use gene elements in the DE optimization process, as illustrated in Fig.5.4.

In addition to generalize our methodology to other video games, we should, as well, apply our methodology to other tasks in game development besides game parameter tuning process. Table 7.1 summarizes our methodology along with other methodologies discussed earlier in subsections 2.2.5.2-2.2.5.4. In (1) Flappy Bird research [23], Isaksen first constructs game space using a survival analysis from the histogram of distance scores obtained from over 106 million simulated game plays. He then analyzes and searches game space for several other applications, e.g. tuning game balance, finding unique yet playable game variants, searching for specific game difficulty, etc. In (3) Legend of Zelda research [19], Gaina tweaks game parameters to search for the diversity of game strategies created by player models.

Looking at the optimized game parameters in (3), in addition to game parameters controlling the game difficulty, Gaina also optimizes some values used to calculate the game scores, i.e. *Score Pickaxe, Score Wall Kill, Time Bonus, Score Gold.* This idea pushes the scope of game balancing to further beyond the control of game difficulty. Although game scores are not directly related to game difficulty, they provide a reward system to help entertain game players. In modern video games, these kind of reward systems are the key to engage the players. This is the reason why they must be considered as a target of our automatic game parameter tuning as well.

7.2 Human Decision Logs for Game Testing

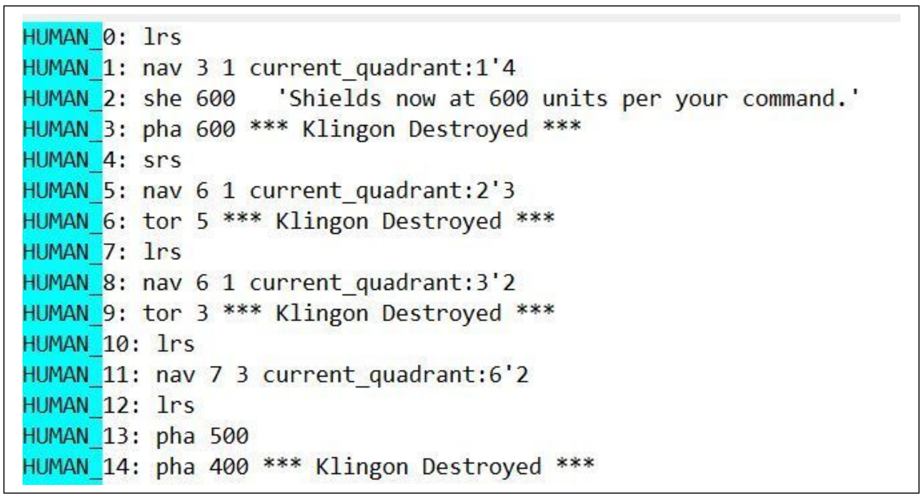
In our framework, we use the FS rule-based player model to simulate a human game player for automatic playtesting. We may extend this concept into a hybrid system that combines both game player models and human game players for some other usages.

We can classify the hybrid systems into two categories:

- **An online hybrid system**: The online hybrid system allows a game human player and a player model to be active at the same time. Both sides may observe or communicate with each other. An example for an online hybrid system is the player assisting system where a player model helps a human player make decisions for the following turns.

- **An offline hybrid system**: The offline hybrid system records command logs for each game decision then uses them for an analysis or evaluation later. An example for an offline evaluation is presented here.

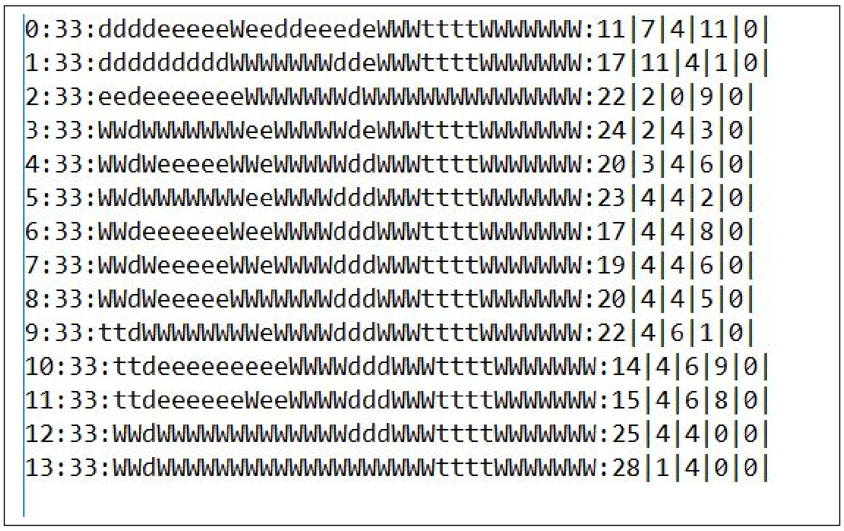
  We use the human player's command logs to evaluate performance of a player model. For each game decision in command logs, we insert a player model into the same playing environment and let the player model plays the game from then on. We document playing results of the player model in each human player's turn. At the end of that game, we have a list of win-loss records of the situation when we replace the human player with the player model.



```
HUMAN_0: lrs
HUMAN_1: nav 3 1 current_quadrant:1'4
HUMAN_2: she 600   'Shields now at 600 units per your command.'
HUMAN_3: pha 600 *** Klingon Destroyed ***
HUMAN_4: srs
HUMAN_5: nav 6 1 current_quadrant:2'3
HUMAN_6: tor 5 *** Klingon Destroyed ***
HUMAN_7: lrs
HUMAN_8: nav 6 1 current_quadrant:3'2
HUMAN_9: tor 3 *** Klingon Destroyed ***
HUMAN_10: lrs
HUMAN_11: nav 7 3 current_quadrant:6'2
HUMAN_12: lrs
HUMAN_13: pha 500
HUMAN_14: pha 400 *** Klingon Destroyed ***
```

Figure 7.1: Sample of human player's command logs while playing a Star Trek game. Each row shows one game command along with the key output response from the game environment.

Figure 7.1 shows the command logs of a human player. For each line of the logs, we put a player model to play the game in place of the human player. At the end of the game, we have an evaluation document like Fig. 7.2 where each character notation represents the win ('W') or loss ('d', 'e', 't') for each turn.



```
0:33:ddddeeeeeWeeddeeedeWWWttttWWWWWWW:11|7|4|11|0|
1:33:dddddddddWWWWWWWddeWWWttttWWWWWWW:17|11|4|1|0|
2:33:eedeeeeeeeWWWWWWdWWWWWWWWWWWWWWWW:22|2|0|9|0|
3:33:WWdWWWWWWWeeWWWWWdeWWWttttWWWWWWW:24|2|4|3|0|
4:33:WWdWeeeeeWWeWWWWWddWWWttttWWWWWWW:20|3|4|6|0|
5:33:WWdWWWWWWWeeWWWWdddWWWttttWWWWWWW:23|4|4|2|0|
6:33:WWdeeeeeeWeeWWWWdddWWWttttWWWWWWW:17|4|4|8|0|
7:33:WWdWeeeeeWWeWWWWdddWWWttttWWWWWWW:19|4|4|6|0|
8:33:WWdWeeeeeWWWWWWWdddWWWttttWWWWWWW:20|4|4|5|0|
9:33:ttdWWWWWWWWeWWWWdddWWWttttWWWWWWW:22|4|6|1|0|
10:33:ttdeeeeeeeeeWWWWdddWWWttttWWWWWWW:14|4|6|9|0|
11:33:ttdeeeeeeWeeWWWWdddWWWttttWWWWWWW:15|4|6|8|0|
12:33:WWdWWWWWWWWWWWWWdddWWWttttWWWWWWW:25|4|4|0|0|
13:33:WWdWWWWWWWWWWWWWWWWWWWttttWWWWWWW:28|1|4|0|0|
```

Figure 7.2: Sample of a player model evaluation from a human player's command logs. Each row shows an evaluation of each turn in a game with the following character notations: d = defeat, e = energy run out, t = time out, W = win. Each small letter notation, except 'W', refers to a lost game.

7.3 Influence of FS Rule Complexity on the Incremental Learning

This small experiment is a spin-off from Chapter 6 for gradual incremental learning. The objective is to observe the effect of the FS rule complexity with different numbers of optimized parameters. We have an assumption that more complicated rules help a player model perform better than simple rules.

For all experiments in our research, in both Chapter 5 and 6, we use simple FS rules as shown in Appendix B. The rules contain nine Boolean and five FS input variables from five modular tables. All FS inputs are binary FS variables, consisting of LOW or HIGH membership levels. Therefore, we have a total of ten FS membership function parameters.

Based on the simple rules, we create the extended FS rules (see Appendix C) with the following modifications, while still maintaining all modular table relationship:

- **Adding one extra table for handling the critical event**: This is shown in Table. C.2. The table handles a critical event when the Enterprise enters the quadrant where the Klingon spaceship exists.

- **Adding two value tables to determine energy usage**: This is an implementation of FS rules to compute a command argument.

- **Adding more FS input variables to the existing tables**: We add more input conditions to the table to tune up the decision details.
- **Expand FS membership level in some input variables**: This is also an FS table tuned-up for additional decision details.

As a result, we now have 8 Boolean and 14 FS input variables in 6 FS modular tables and 2 FS value tables. Among 14 FS variables, 12 are binary (LOW/HIGH) FS inputs and the other 2 are tertiary (LOW/MEDIUM/HIGH) FS inputs. Therefore, the number of FS membership function parameters are extended to 30 parameters. Additional information on extended rules can be found in subsection 4.5.3.2.

7.3.1 Experimental Setups

The setups are the same as the experiment done on adaptive-interval incremental learning in subsection 6.3.3. The only difference is we use the extended FS rules in place of the simple ones. We run three experiments for a standard DE-optimized player model (without incremental learning) as well as an incremental learning DE-optimized player model with stable-fitness interval at 5 and 10 DE generations.

7.3.2 Experimental Results

Figure 7.3 illustrates the improvement result of the player model with the extended FS rules from this experiment, along with the performance of the player model with the simple rules from subsection 6.3.3.2. The results show that there is almost no improvement in the player model with the extended FS rules in adaptive-interval incremental learning.

7.3.3 Discussion

The incremental learning player models show no sign of improvement when using the extended FS rules, compared to the ones with the simple rules. However, all models with extended rules, even the standard one without incremental learning, outperform all models with the simple rules. This may indicate that the extended rules with more optimized parameters do improve the performance of a player model. Nevertheless, additional experiment data are required to confirm this assumption. As extended rules are originally based on the simple rules, they share some common characteristics. Both FS rules may probably have similar parameter landscapes with only some difference in the boundary details. In that case, we may need to

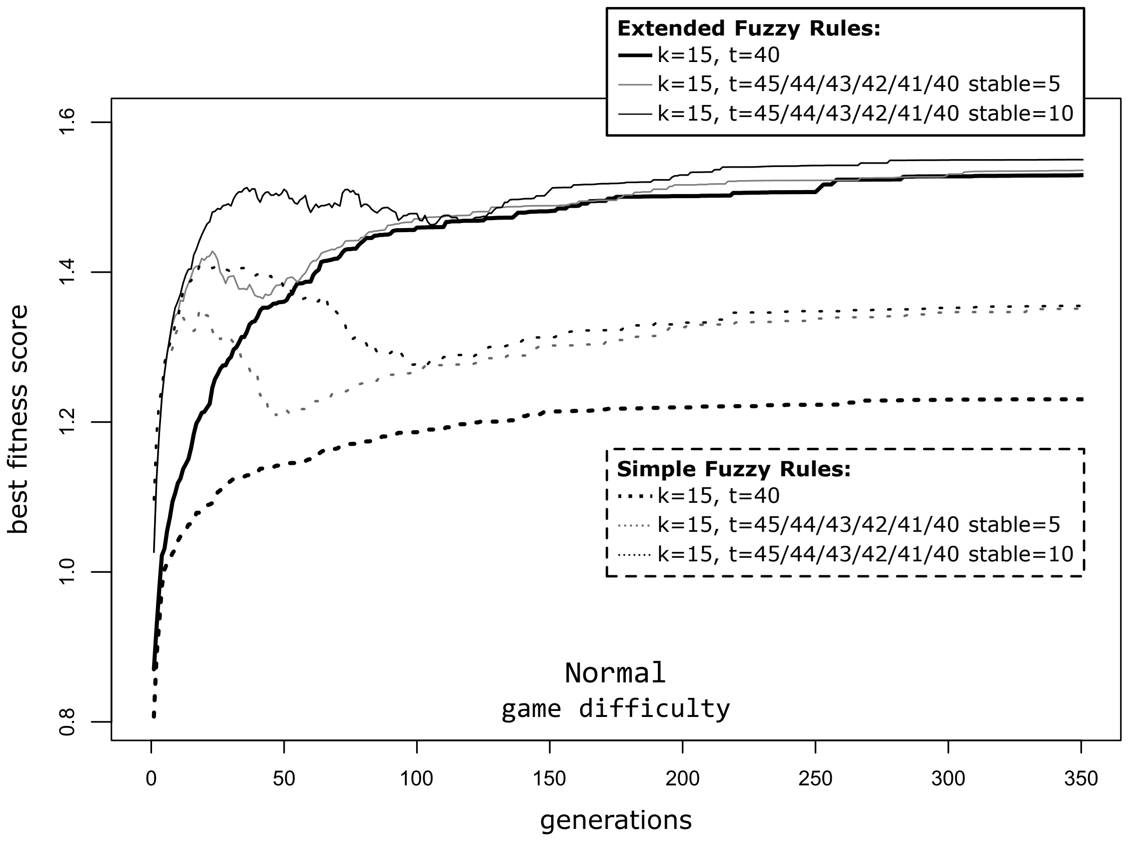


Figure 7.3: Best fitness score comparison between a group of player models optimized by simple fuzzy rules and by extended fuzzy rules. Each group of player models consists of a simple DE-optimized player model and three DE-optimized player models with gradually adaptive incremental learning of game difficulty. The game difficulty increases by adaptively reducing one Stardate when the best fitness scores are continuously stable for 5, 10, and 25 DE generations. The game time reduces from 45 Stardates in easy games to 40 Stardates in normal games.

specifically investigate each rule to see how many times it is called for the game decisions. A comparison analysis of the activated rules between both sets may then give us further ideas.

7.4 Chapter Summary

In our game tuning framework, the conceptual idea toward automatic tuning for a wider range of game players is the coevolution between the game and the multi-skilled player model. The evaluation of a player model takes on an important role in classifying gaming skills of a player model as precisely as possible. There are many approaches in the player model evaluation. While the player model evaluation using DE fitness functions is an evaluation from a game developer's point of view, the player model evaluation using a human player's command logs is an evaluation via the player's actual game decisions. Evaluating the player model accurately is another key success factor of automatic game tuning.

Another important issue in the coevolution process is that it works efficiently when the abilities of both sides are not far apart. If one side is much stronger than the other side, the coevolution is not likely to take place between them. This is because the much stronger party will always win, giving the weaker one no chances to fight back. In such a case, the coevolution process cannot be successful. Therefore, improving the performance of the player model to be competitive with the game is another point of concern.

Chapter 8

Conclusion and Future Works

In this research, we propose a framework for automatic game parameter tuning to search for game parameters that are suitable for a broader range of game players. The framework consists of an FS rule-based, EC-optimized game player model along with a coevolutionary algorithm (CEA). The main idea is to create simulated game players with various gaming skills from the player model and use them to fine-tune game parameters that match the diversity of game-playing abilities. We implement the framework with a turn-based strategy (TBS) game called Star Trek.

Our first experiments in the research show that the player model is able to compete very well with the game in many levels of game difficulty, thanks to the DE optimization process. In addition to improving the model performance, DE optimization also helps to ease the burden of specifying the FS membership function parameters. We still require game developers to integrate their expertise in the game to create decision rules for an FS rule-based system. Their involvement helps to guide game decisions to the direction required.

With the gradual incremental learning in the second experiment, our DE-optimized player model exhibits some improvement. However, the small yet robust improvement from the manual game parameter adjustment signifies the potential power of mutual evolution process between Star Trek game and the DE-optimized player model.

Nevertheless, the automatic game parameter tuning using CEA approach, which is the final step to complete our framework, is still waiting for our investigation as a work in progress.

8.1 Future Works

The high-priority work is to verify whether CEA approach is able to coevolve game parameters with multi-skilled player models and, if so, how well the approach performs. In addition, there are a number of other related ideas worth exploring, which we found during our path toward the goal of automatic game tuning research. Some of them are:

- **The extension of the FS rule-based player model to other video games**: We use an FS rule-based decision-making system, guided by game developers' expertise, to make decisions in a player model for TBS games. Focusing on logical decision makings, the nature of strategy games commonly matches the characteristics of the FS rule-based system. With new FS rules specifically created for the game, we believe that we can extend our approach to other TSB games smoothly. However, the solid proof to show that our methodology is truly valid for TBS games in general, besides Star Trek game in our simulation test bed, is another major work requiring a more thorough investigation.

  For general strategy games, including real-time strategy (RTS) games, the issues of game complexity and computation time are considered major concerns that require further study. However, an FS rule-based decision-making system may not be an efficient controller for a player model in non-strategy games, e.g. action games, puzzle games, role-playing games, etc.

- **The in-depth study on the FS rules for game decisions**: Our system requires a skilled player to provide his or her video game expertise in the form of FS rules. Different sets of rules from many players should provide the player model a variety of game-playing abilities. We use a set of simple FS rules for most of our experiments. One exception is the pilot experiment on the influence of FS rule complexity in section 7.3. We believe that the more complex the rules are, the better the player model's performance becomes. The detailed study in this topic may have a strong effect on an EC approach used in our framework.

- **The practical use of controlling policies in CEA**: The controlling policies in CEA represent a flexible tool for the game developers to bias the coevolution toward any required target. However, the policy is simply a guideline for the desired mutual evolution. The idea of CEA is based on the concept of dynamic interaction. Therefore, the outcome of the policy may not be the same as expected in such a dynamic environment. We expect additional investigations on the implementation of the CEA policies that can achieve the practical coevolution results.

8.2 Limitations

Despite our fine experimental results, there are a few restrictions and drawbacks in the approach of our framework. The major limitations are:

- **The limitation of FS rule-based decision-making system**: Our player model plays a TBS game by issuing valid game commands that correspond to game actions decided by the model. However, a game command consists of an action command and command arguments, which specify how to perform the action. While FS rules are suitable for determining game actions, they may not be capable of computing some command arguments. In many cases, a direct mathematical computation of numerical arguments provides an easier implementation and more accurate results than a rule-based system does. For example, the Star Trek game requires a torpedo direction as a command argument when firing a torpedo. In this case, the geometric calculation of the direction between the Enterprise and the Klingon spaceship ensures a success of the attack. It is not straightforward to use a rule-based system in such a case. Therefore, FS rule-based decision-making system has some restrictions as a controller in a player model.

- **The limitation of EC algorithms**: An EC algorithm is a population-based algorithm. The parameter search is done by an individual population, and the search results improve over the generations of searches One of the problems with the approach, especially when playing a game with no time limit or with open endings, is its computation time spent in the search. An EC approach may not be efficient when playing real-time games, where a speed of actions is the decisive factor.

- **The difficulty to create an efficient player model**: The goal of this research is to reduce the developing time in a video game production by automating the game tuning process. Our approach, as well as other approaches discussed earlier in related research, uses a player model as simulated game players in a playtesting process. However, it is not easy to create a player model that is efficient enough for an automated playtesting. It may end up spending more time on creating an efficient player model than tuning the game parameters manually.

This drawback relies heavily on the fact that a player model is highly game-specific. However, as suggested by many papers [31] [19] [26], we can employ ready-made player models provided by GVG-AI, an autonomous game agent competition. These models are a general-purposed agent that learns to play unforeseen video games in the competition. This will hugely reduce the time to create a player model for certain games in which the autonomous game agent excels with.

- **The difficulty to apply an academic research to practical uses**: The most important issue in automatic game tuning is to apply the proposed methodology in a real practice. There is quite a big obstacle to put knowledge gained from the academic game research into practice for the game industry [64]. With a tight developing schedule and competitive market, it is difficult for the industry to adopt a new approach that affects their development workflow. A new methodology from an academic research may not be suitable for the game industry, without extensive proofs of practical successes.

8.3 Conclusion

Automatic game parameter tuning is a research area in video games that comprises many overlapping research fields. One is the area of automation in video games that includes an automatic creation of game rules and contents, an automatic game playing, etc. This research area is the most popular one in video game researches. Another associated area is the AI-assisted game design which develops supporting tools for game designers or game developers. This area requires a high level of control from the game developers to guide the development process toward their design plan.

However, automatic game parameter tuning is still a new research area in the field of Computational Intelligence in Games. Unlike other game researches which emphasize on the game creation or game playing, research in game tuning involves a comprehensive exploration of game parameter space in an existing video game. It had been an unexplored research field until the past few years when a number of game researchers started to get interested in this area.

To the best of our knowledge, the proposed methodology is the first attempt in automatic parameter tuning for a turn-based strategy video game. In the framework, we apply a fuzzy rule-based player model with evolutionary computation techniques for a game tuning process. Although the valid results are still unclear due to the lack of supports in a coevolutionary algorithm, the framework has a high potential to serve its purpose well. With plenty of aspects waiting to be explored in this area, we hope that our work may be useful for researchers in the area of video game tuning, which is still at the beginning of its progress.

Bibliography

[1] G. Andrade, G. Ramalho, H. Santana, and V. Corruble. Automatic Computer Game Balancing: A Reinforcement Learning Approach. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 1111–1112, Utrecht, Netherlands, 2005.

[2] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[3] C. Browne. *Evolutionary Game Design.* Springer, 2011.

[4] A. B. Cardona, J. Togelius, and M. J. Nelson. Competitive coevolution in Ms. Pac-Man. In *IEEE Congress on Evolutionary Computation (CEC 2013)*, pp. 1403–1410, 2013.

[5] N. Casas. A review of landmark articles in the field of co-evolutionary computing. 2015. arXiv:1506.05082 [cs.NE].

[6] J. Chen. Flow in Games (and everything else). *Communications of the ACM*, 50(4):31–34, 2007.

[7] W. E. Combs. The Combs method for rapid inference. In E. Cox and M. O'Hagan, editors, *The Fuzzy Systems Handbook, Second Edition: A Practitioner's Guide to Building, Using, and Maintaining Fuzzy Systems*, pp. 659–680. AP Professional, 1998.

[8] M. Črepinšek, S.-H. Liu, and M. Mernik. Exploration and exploitation in evolutionary algorithms. *ACM Computing Surveys*, 45(3):1–33, 2013.

[9] S. Das, S. S. Mullick, and P. N. Suganthan. Recent advances in differential evolution - An updated survey. *Swarm and Evolutionary Computation*, 27:1–30, 2016.

[10] S. Das and P. N. Suganthan. Differential evolution: A survey of the state-of-the-art. *IEEE Transactions on Evolutionary Computation*, 15(1):4–31, 2011.

[11] A. DaSilva. 2016 Top Markets Report Media and Entertainment. Technical report, International Trade Administration, U.S. Department of Commerce, 2016.

[12] P. David. Flights of Fancy with the Enterprise. *Byte Magazine*, 2(3):106–113, 1977.

[13] S. A. Dias, S. Alves, and D. Yuka. Star Trek ゲームプレーヤ意思決定モデルの進化：Evolving a Human Player Model for the Star Trek Game. In *Joint meeting of 2nd Evolutionary Computation Meeting and 6th Evolutionary Computation Frontier Meeting*, pp. 112–117, Toyonaka, Japan, 2012. (in Japanese).

[14] S. Egenfeldt-Nielsen, J. H. Smith, and S. P. Tosca. *Understanding video games: The essential introduction.* Routledge, New York, USA, 2010.

[15] A. P. Engelbrecht. *Computational intelligence: An introduction.* Wiley, 2nd edition, 2007.

[16] N. Ensmenger. Is chess the drosophila of artificial intelligence? a social history of an algorithm. *Social Studies of Science*, 42(1):5–30, 2012.

[17] G. S. Etchebehere, P. Mackenzie, and F. D. Computac. L-Systems and Procedural Generation of Virtual Game Maze Sceneries. In *Proceedings of SBGames 2017*, pp. 602–605, Curitiba, Brazil, 2017.

[18] M. Fukuda. *Bit Generation 2000: TV Game Exhibition.* Contemporary Art Gallery of Art Tower Mito and Kobe Fashion Museum, Mito, Japan, 2000. (in Japanese/English).

[19] R. Gaina, R. Volkovas, C. González Díaz, and R. Davidson. Automatic Game Tuning for Strategic Diversity. In *Computer Science and Electronic Engineering Conference (CEEC 2017)*, pp. 195–200, Colchester, UK, 2017.

[20] F. Herrera and M. Lozano. *Fuzzy Evolutionary Algorithms and Genetic Fuzzy Systems: A Positive Collaboration between Evolutionary Algorithms and Fuzzy Systems Computational Intelligence*. Springer, 2009.

[21] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D: Nonlinear Phenomena*, 42(1-3):228–234, 1990.

[22] A. Isaksen, D. Gopstein, J. Togelius, and A. Nealen. Discovering Unique Game Variants. In *Computational Creativity and Games Workshop at The 6th International Conference on Computational Creativity (ICCC 2015)*, Park City, Utah, USA, 2015.

[23] A. Isaksen, D. Gopstein, J. Togelius, and A. Nealen. Exploring Game Space of Minimal Action Games via Parameter Tuning and Survival Analysis. *IEEE Transactions on Computational Intelligence and AI in Games*, pp. 1–13, 2017. (in print).

[24] L. Johnson, G. N. Yannakakis, and J. Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pp. 1–4, Monterey, CA, USA, 2010.

[25] G. Kasparov. The Chess Master and the Computer. *The New York Review of Books.* 57(2), February 11, 2010.

[26] K. Kunanusont, R. D. Gaina, J. Liu, D. Perez-Liebana, and S. M. Lucas. The N-Tuple bandit evolutionary algorithm for automatic game improvement. In *IEEE Congress on Evolutionary Computation (CEC 2017)*, pp. 2201–2208, San Sebastián, Spain, 2017.

[27] F. Lantz, A. Isaksen, A. Jaffe, A. Nealen, and J. Togelius. Depth in Strategic Games. In *What's Next in AI in Games Workshop in The 31st AAAI Conference on Artificial Intelligence*, San Francisco, USA, 2017.

[28] M. A. Lee and H. Takagi. A Framework for Studying the Effects of Dynamic Crossover, Mutation, and Population Sizing in Genetic Algorithms. In T. Furuhashi, editor, *Advances in Fuzzy Logic, Neural Networks and Genetic Algorithms*, chapter 8, pp. 111–126. Springer-Verlag, 1995.

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

[30] A. Liapis, G. N. Yannakakis, and J. Togelius. Computational Game Creativity. In *Proceedings of the 5th International Conference on Computational Creativity*, pp. 46–53, Ljubljana, Slovenia, 2014.

[31] J. Liu, J. Togelius, D. Perez-Liebana, and S. M. Lucas. Evolving Game Skill-Depth using General Video Game AI agents. In *IEEE Congress on Evolutionary Computation (CEC 2017)*, pp. 2299–2307, San Sebastián, Spain, 2017.

[32] S. M. Lucas. Computational intelligence and AI in games: A New IEEE transactions. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):1–3, 2009.

[33] S. M. Lucas and G. Kendall. Evolutionary computation and games. *IEEE Computational Intelligence Magazine*, 1(1):10–18, 2006.

[34] E. Mamdani. Application of fuzzy algorithms for control of simple dynamic plant. *Proceedings of the Institution of Electrical Engineers*, 121(12):1585, 1974.

[35] I. Millington and J. D. Funge. *Artificial Intelligence for Games.* CRC Press, 2009.

[36] V. Mnih, K. Kavukcuoglu, D. Silver, et al. Playing Atari with Deep Reinforcement Learning. In *Deep Learning Workshop at Neural Information Processing Systems (NIPS 2013)*, pp. 1–9, Lake Tahoe, USA, 2013.

[37] M. J. Nelson and M. Mateas. Towards Automated Game Design. In *Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence*, pp. 626–637, Rome, Italy, 2007.

[38] Nielsen Games. Games 360 U.S. Report 2017. Technical report, The Nielsen Company, 2017.

[39] S. K. Park and K. W. Miller. Random Number Generators: good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[40] C. A. Peña-Reyes. *Coevolutionary Fuzzy Modeling.* Springer, 2004.

[41] Piroyan. Star Trek 宇宙大作戦, 2007. http://lablog.piroyan.com/?e=41, Last accessed on 2018-06-06. (in Japanese).

[42] M. Potter and K. D. Jong. A Cooperative Coevolutionary Approach to Function Optimization. In *International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature*, pp. 249 – 257, Jerusalem, Israel, 1994.

[43] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[44] K. V. Price, R. M. Storn, and J. A. Lampinen. *Differential Evolution - A Practical Approach to Global Optimization*. Springer-Verlag, 2005.

[45] M. O. Riedl and A. Zook. AI for Game Production. In *IEEE Conference on Computatonal Intelligence and Games*, pp. 1–8, Niagara Falls, Canada, 2013.

[46] J. Schaeffer, N. Burch, Y. Björnsson, et al. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[47] J. Schell. *The Art of Game Design*. CRC Press, Boca Raton, FL, 2nd edition, 2014.

[48] C. P. Schultz, R. Bryant, and T. Langdell. *Game Testing All in One*. Thomson Course Technology PTR, Boston, MA, 2005.

[49] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games*. Springer, 2016.

[50] D. Silver, A. Huang, C. J. Maddison, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

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

[52] R. Storn and K. Price. Differential Evolution - A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[53] M. Sugeno and G. T. Kang. Structure identification of fuzzy model. *Fuzzy Sets and Systems*, 28(1):15–33, 1988.

[54] A. Summerville, S. Snodgrass, M. Guzdial, et al. Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games*, pp. 1–15, 2018. (in print).

[55] H. Takagi. Introduction to Fuzzy Systems , Neural Networks , and Genetic Algorithms. In D. Ruan, editor, *Intelligent Hybrid Systems: Fuzzy Logic, Neural Networks, and Genetic Algorithms*, chapter 1, pp. 3–33. Springer, Boston, MA, 1997.

[56] T. Takagi and M. Sugeno. Fuzzy identification of systems and its applications to modeling and control. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-15(1):116–132, 1985.

[57] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG 2008)*, pp. 111–118, Perth, WA, Australia, 2008.

[58] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[59] B. Vallade, A. David, and T. Nakashima. Three layers framework concept for adjustable artificial intelligence. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 19(6):867–879, 2015.

[60] V. Vorachart and H. Takagi. Evolving Fuzzy Logic Rule-Based Game Player Model for Game Development. *International Journal of Innovative Computing*, 13(6):1941–1951, 2017.

[61] R. P. Wiegand, W. C. Liles, and K. A. De Jong. An Empirical Analysis of Collaboration Methods in Cooperative Coevolutionary Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pp. 1235–1242, San Francisco, USA, 2001.

[62] A. P. Witkin. Scale-space filtering. In *International Joint Conference on Artificial Intelligence*, vol. 2, pp. 1019–1022, Karlsruhe, Germany, 1983.

[63] G. N. Yannakakis and J. Togelius. A Panorama of Artificial and Computational Intelligence in Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):317–335, 2015.

[64] G. N. Yannakakis and J. Togelius. *Artificial Intelligence and Games*. Springer, 2018.

[65] L. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1965.

Acknowledgements

I would like to express my very great appreciation to Professor Hideyuki Takagi, my advisor, for his valuable teachings, patient guidance, and demonstrating in all kinds of ideas and actions, both academically and socially. As a fine researcher and an excellent educator, he has been and will always be a role model in my academic life and teaching career.

I would also like to thank my three referees: Professors Hideyuki Takagi, Mario Köppen, and Osamu Maruyama for their precious time and constructive comments on this dissertation.

My special thanks are extended to all my laboratory colleagues from October 2014 to September 2018: Hayakawa Masahiro, Li Tian, Xie XiangYu, Odd Rune LYKKEBØ, Ikeda Keisuke, Horie Takahiro, Matsukuma Takafumi, Motomura Shun'nosuke, Tanaka Chikahiko, Lai WeiQiang, Li YuHao, Ooishi Wataru, Moribe Honoka, Xiong YanRan, especially to Yu Jun and Takuma-kun. タクマさん、「学芸は長し、生涯は短し」。

My life in Japan would be more difficult without the help from all staff in Support Center for international students, starting from my first contact to Kyushu University until my very last day in Ohashi. In addition, I really appreciate the assistance of Ms.Sakiko Tanaka of Academic Affairs for helping me fill out all the paper works in Japanese.

Funding and support provided during my doctoral study in Kyushu University by the Royal Thai Government Scholarship and Walailak University are always greatly appreciated. I am particularly grateful for the assistance and facilities given by the Office of Education Affairs in Royal Thai Embassy in Tokyo.

Finally, I wish to thank all relatives in Warrachart, Petchmunee, and Preechawong families for all forms of support and supplies, here, there, and everywhere. Special thanks should be given to my elder brothers and their family members: P'A & P'Tle and P'Boy for always having faith in me; my cousin: P'Nid for her helpful proofreading; Hong Sopida & Beckham - *the beloved ones* - for her caretaking, home cooking, and sweet smiling in Shiobaru; and, last but not least, my Pa and Mama for their deep and sincere love in me.

Varunyu Vorachart

Shiobaru, Fukuoka
August 21, 2018

Publication

This is a journal publication during my doctoral research.

Chapter 5: Evolving Fuzzy Logic Rule-based Player Model

1. **V. Vorachart** and H. Takagi. *"Evolving Fuzzy Logic Rule-Based Game Player Model for Game Development"*. International Journal of Innovative Computing, 13(6):1941-1951, 2017.

Appendix A

Star Trek Game

A.1 Overview

Star Trek game is a turn-based strategy game originating from a classic American television series, Star Trek. Debuted in 1966, the Star Trek TV series has become popular since then. The series continued its showings for a long time until now with many remakes and reruns. The original series portrayed the adventures of Captain James T. Kirk and his crew on the Starship USS Enterprise. They travel to accomplish special missions throughout the galaxy. During 1960s, the series characterized an alien humanoid species, called the Klingons, as a recurring antagonist for a mankind. However, the Klingons became a close ally in the later series, on and off.

This science fiction TV series has been an addict worldwide due to its hopeful tales of the future that involve space adventures and team friendships in the USS Enterprise. A large number of media franchises and merchandises have been released, including motion pictures, books, comics, magazines, card games, board games, video games, as well as virtual reality games.

The Star Trek game used in our experiment originates from Mike Mayfield's *Trek* computer game. Trek was written in BASIC programming language in 1971. It is a text-based computer game that personates a player as a captain of the USS Enterprise on a mission to search and destroy all invading Klingon spaceships. Our source code is a free software port of *Super Star Trek* [12], the most popular version of 1971 Trek game. The code is written in C by Piroyan for MSX-DOS operating system in 2007 [41]. For simplicity, we refer to our simulated game in this dissertation as *Star Trek game*.

A.2 Gameplay

A player controls the Starship Enterprise to survey the 8×8 quadrants in a galaxy. The objective is to destroy all Klingon spaceships within a given game time. Two kinds of weapons can be used for an attack: photon torpedo and phaser. The player has to manage the energy for navigating, offensive, and defensive strategies. The energy can be recharged at one of the Starbases, which are located throughout the galaxy.

A.2.1 Winning and Losing Conditions

The unit of game time in Star Trek game is called Stardate. One Stardate is close to a period of time required in travelling across one quadrant. Stardate advances after a player's turn.

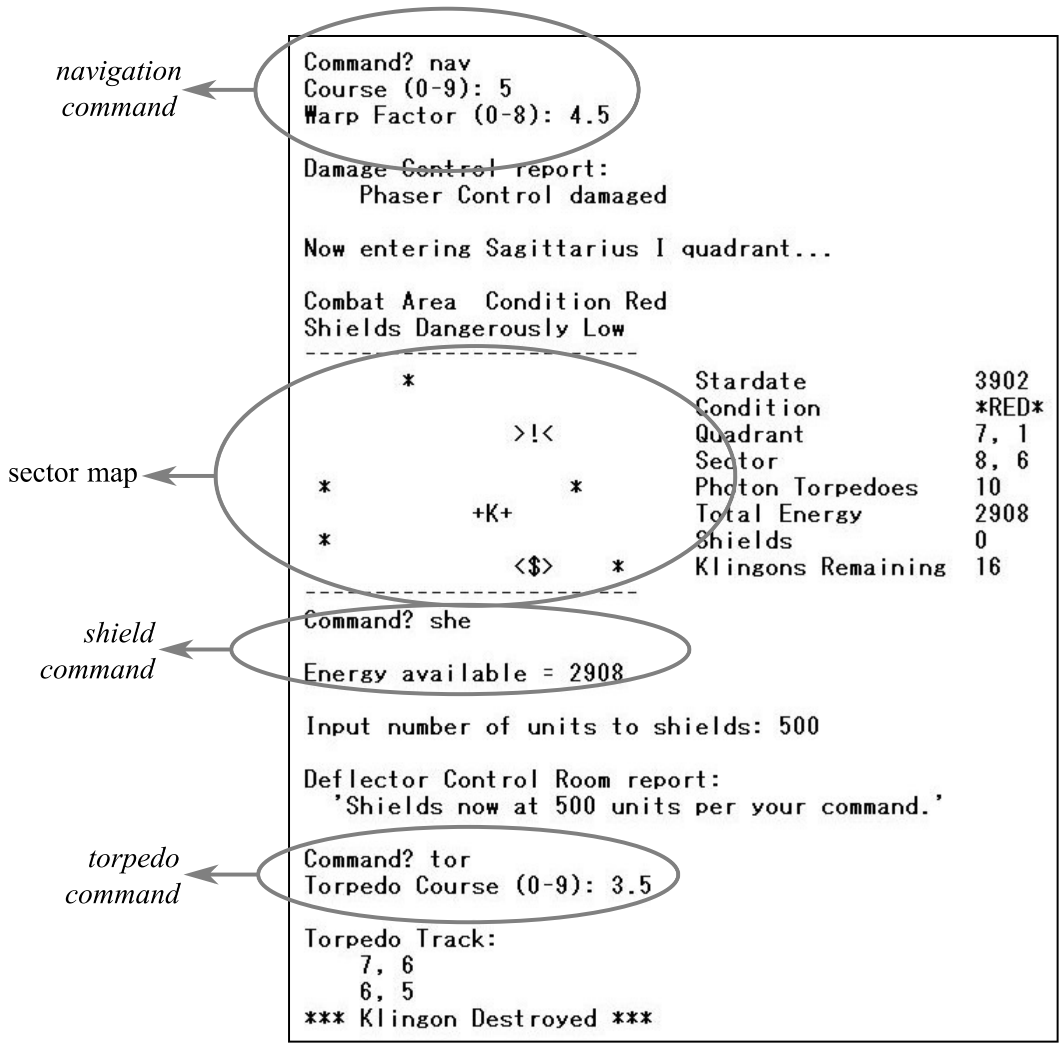
The player wins the game when all Klingon spaceships are destroyed within the given game time. Otherwise, the player loses the game under three circumstances: being defeated, running out of time, running out of energy.

A.2.2 Energy Management

Initially, the Starship Enterprise contains 3,000 units of energy and 10 photon torpedoes. All operating devices of the Enterprise are in good conditions.

The USS Enterprise uses its energy for three activities: navigation, phaser attack, and defensive shield. The inter-quadrant navigation at a high warping speed consumes more energy than at a low speed. The Enterprise can attack the Klingons with the phaser energy. This weapon provides the most powerful yet energy-intensive attack. The defensive shield is the shared energy that must be set explicitly to protect the Enterprise. The shield energy decreases under the Klingon's attack. The game is over when the shield energy is below zero. However, the Enterprise can recharge its energy at the Starbases.

The docking operation between the Enterprise and the Starbase takes place when the Enterprise moves into a Starbase's vicinity. The Starbase recharges the Enterprise's energy automatically to its maximum capacity of 3,000 units. Additionally, the Starbase also refills the torpedoes instantly back to its initial amount of 10 units. Nevertheless, it can repair the Enterprise's damaged devices only at the explicit request.



```
Command? nav
Course (0-9): 5
Warp Factor (0-8): 4.5

Damage Control report:
    Phaser Control damaged

Now entering Sagittarius I quadrant...

Combat Area  Condition Red
Shields Dangerously Low
------------------------
       *                    Stardate            3902
                            Condition           *RED*
              >!<           Quadrant            7, 1
                            Sector              8, 6
 *                 *        Photon Torpedoes    10
           +K+              Total Energy        2908
 *                          Shields             0
              <$>    *      Klingons Remaining  16
------------------------
Command? she

Energy available = 2908

Input number of units to shields: 500

Deflector Control Room report:
  'Shields now at 500 units per your command.'

Command? tor
Torpedo Course (0-9): 3.5

Torpedo Track:
    7, 6
    6, 5
*** Klingon Destroyed ***
```

Figure A.1: A screen capture of Star Trek game showing a player destroying a Klingon spaceship with a photon torpedo.

A.2.3 Game Objects and Sector Map

The Enterprise navigates among the quadrants to search for Klingon spaceships. A quadrant is a space of 8 × 8 sectors. Each sector is either empty or occupied by a game object. A sector map, as shown in the middle of Fig.A.1, displays all game objects located in the current quadrant. Four types of game objects and their detailed descriptions are listed below.

- **Star:** The symbol $*$ in a sector map represents a star. A star is stationary inside a quadrant. It acts as an obstruction for the Enterprise's navigator. It also blocks the Enterprise's photon torpedo when its position is located on the torpedo path. A quadrant must contain at least one star at the minimum.

- **Starbase:** The symbol <!> in a sector map represents a Starbase. In a galaxy, the number of Starbases is very limited. Its location is unknown originally. Only one Starbase can reside in a quadrant. It stays still in the quadrant. Nevertheless, both Starbases and Klingon spaceships may reside in the same quadrant.

- **Klingon spaceship:** The symbol $+K+$ in a sector map represents a Klingon spaceship. A maximum of three Klingon spaceships can reside in the same quadrant. The energy of each spaceship is different and unknown to a player. The Klingons can only move inside its quadrant. Unlike the Enterprise, however, it can move and fire a weapon in the same turn.

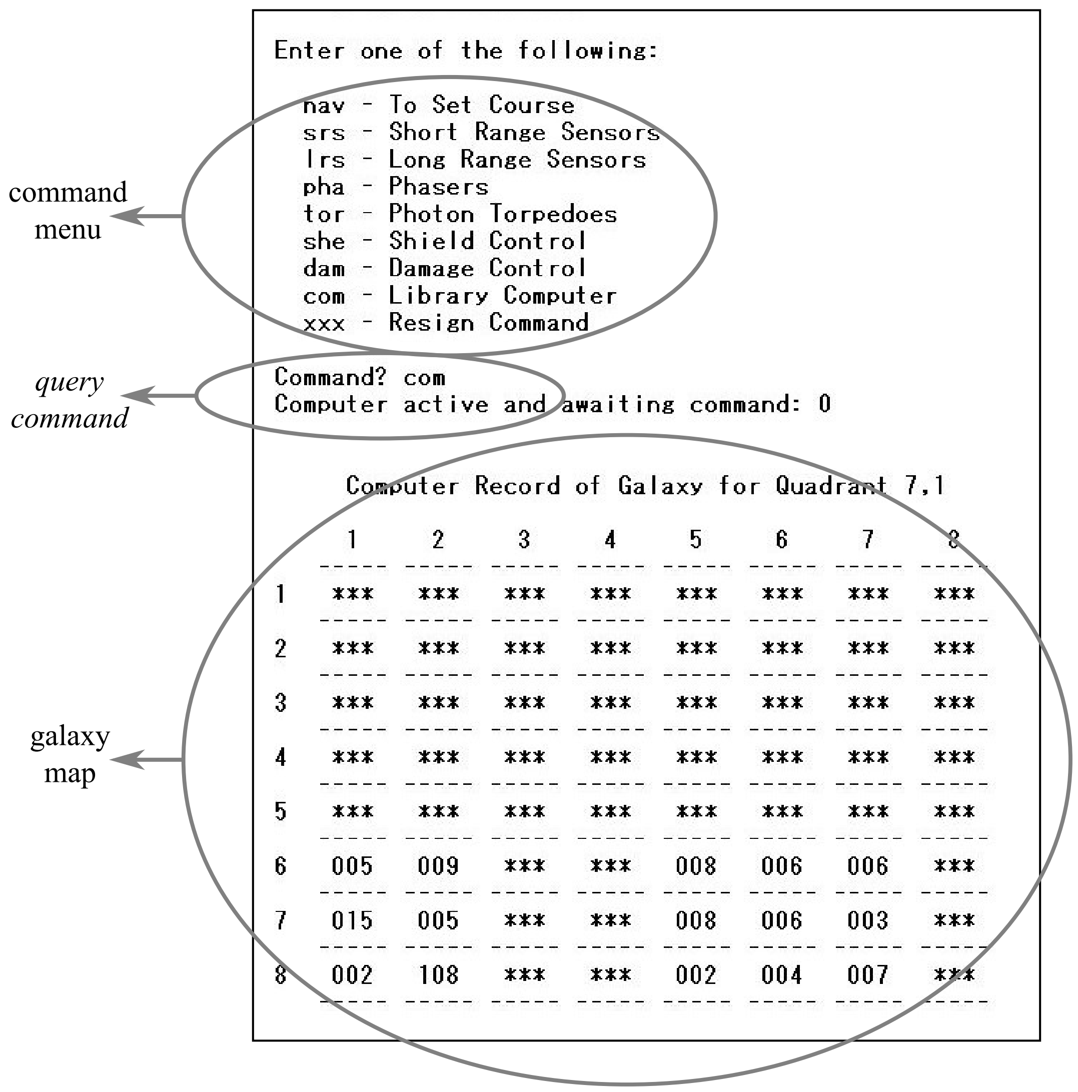
- **Starship Enterprise:** The symbol <$> in a sector map represents the Starship Enterprise. There is only one Starship Enterprise in the game. Its initial position is randomly located.

  At the beginning of the game, the Enterprise contains 3,000 units of energy and 10 photon torpedoes, with all devices in good conditions. The information in the Library-Computer device is blank, including the galaxy map.

A.2.4 Galaxy Map

When a game starts, all game objects are distributed randomly throughout the galaxy. The number of each game object is determined according to the level of game difficulty and the game ID. Their locations are then assigned to each quadrant. Unlike the Enterprise, which can move freely among the quadrants, all other objects cannot move out of a quadrant. However, when the Enterprise enters a new quadrant, game objects in that quadrant rearrange their locations anew. Inside a quadrant, all stars and Starbase cannot move while Klingon spaceships can move occasionally only in their turn.

The galaxy map, as shown in the bottom of Fig. A.2, displays the record of a long-range sensor (LRS) device. The LRS device shows nine numerical values representing the number of the game objects in the current quadrant and eight immediate surroundings. Each number has three digits. Each digit denotes to the number of Klingon spaceships, Starbase, and star objects in a quadrant, respectively. For example, the number of 015 represents no Klingons, one Starbase, and five stars, correspondingly. The number changes when the Klingon spaceship (or the Starbase!) is destroyed.



```
Enter one of the following:

  nav - To Set Course
  srs - Short Range Sensors
  lrs - Long Range Sensors
  pha - Phasers
  tor - Photon Torpedoes
  she - Shield Control
  dam - Damage Control
  com - Library Computer
  xxx - Resign Command

Command? com
Computer active and awaiting command: 0


     Computer Record of Galaxy for Quadrant 7,1

      1     2     3     4     5     6     7     8
    ----- ----- ----- ----- ----- ----- ----- -----
1    ***   ***   ***   ***   ***   ***   ***   ***
    ----- ----- ----- ----- ----- ----- ----- -----
2    ***   ***   ***   ***   ***   ***   ***   ***
    ----- ----- ----- ----- ----- ----- ----- -----
3    ***   ***   ***   ***   ***   ***   ***   ***
    ----- ----- ----- ----- ----- ----- ----- -----
4    ***   ***   ***   ***   ***   ***   ***   ***
    ----- ----- ----- ----- ----- ----- ----- -----
5    ***   ***   ***   ***   ***   ***   ***   ***
    ----- ----- ----- ----- ----- ----- ----- -----
6    005   009   ***   ***   008   006   006   ***
    ----- ----- ----- ----- ----- ----- ----- -----
7    015   005   ***   ***   008   006   003   ***
    ----- ----- ----- ----- ----- ----- ----- -----
8    002   108   ***   ***   002   004   007   ***
    ----- ----- ----- ----- ----- ----- ----- -----
```

Figure A.2: A screen capture of Star Trek game showing the command menu and the entire galaxy map.

A.2.5 The Battle with The Klingons

When the Enterprise enters a Klingon-occupied quadrant, the game shows a warning of combat area with ∗*RED*∗ condition. The top of Fig. A.1 displays this warning as a result of the navigation command. The battle starts with the Enterprise's turn then followed by the Klingons' turn alternately. Escaping the combat area in the middle of the fight results to a follow-up attack from the remaining Klingons.

- **Offensive Strategy:** There are two kinds of weapons to attack the Klingons.
  - **Phaser Energy**: This weapon is omni-directional. It can, therefore, destroy many Klingon spaceships in one blaze. Firing the phaser consumes the Enterprise's energy. A player must specify the amount of energy

needed to fire the phaser. The amount of energy required to destroy a Klingon is proportional to the number of Klingons in the quadrant and the distance of the Klingon away from the Enterprise. If the firing energy is low, the attack will only damage the Klingon and lessen its energy. This reduces its strength to fight back.

  - **Photon Torpedo**: This is a uni-directional weapon that launches straightly in a specified direction. Hence, it can destroy only one Klingon spaceship at a time. The Enterprise contains 10 torpedoes, initially, so the number of torpedo attacks is limited. Unlike the phaser energy, which nothing can hamper, the torpedo attack becomes ineffective when its launched path is blocked by a star. The Enterprise refills its torpedoes automatically at the Starbases.

- **Defensive Strategy:** The Enterprise has to set its shield energy to prevent a defeat. In the battlefield, this shield energy counteracts with the Klingons' attack energy to protect the Enterprise. When the shield energy goes below zero, the Klingons win the battle and the player loses the game.

  Shield energy is a part of the Enterprise's energy. The Enterprise has to lower its shield energy when the energy is not enough for other activities, i.e. navigation or phaser attack. Nevertheless, the shield energy cannot be altered when the shield device is damaged.

In case that the attack fails, the game output will provide enough information to help the player examine the errors. For example, the torpedo track, shown in the bottom of Fig. A.2, displays the torpedo path sector by sector. The player then realizes how far the torpedo misses the target according to this information.

Furthermore, the game output displays the attack information from the Klingons, e.g. the amount of remaining energy and the position of the attacking spaceship. The player may use this information to figure out the plausible sector map, rather than fighting blindly when the SRS device is inoperative.

A.2.6 Damaging Devices

Once in a while, in a battle or a navigation, some of the Enterprise's devices may become damaged. The damaged device is inoperative and needs a repair. With a regular repair process, when the game time passes, the device becomes operative again automatically. The severity of the damage dictates the time period required for the repair process. Alternatively, a Starbase also provides a rapid repair service. It instantly repairs all damaged devices at the expense of some Stardates.

A.3 Game Commands

The command inputs in Star Trek game are simple. A player enters a command name which consists of three characters. Each command has different numbers of its required arguments, from zero up to two arguments. All command arguments in Star Trek game are numerical values, either a real number or an integer number.

There are two major types of game commands in Star Trek game: action commands and query commands. While the action commands control various activities of the Starship Enterprise, the query commands inquire the Enterprise's status as well as the Library-Computer device's report. Certain commands can perform double tasks as an action and a query for different situations.

Star Trek game requests a new command with the *Command?* keyword, as illustrated in the top part of Fig. A.2. In addition, the game displays a command menu listing all nine commands with brief descriptions. Table A.1 lists all commands in Star Trek game along with the required arguments. The following subsections explain each command in more details.

A.3.1 Action Commands

To control the Starship Enterprise, a player issues one of the following five action commands:

1. ***nav*** command sets a direction and a distance of a navigation. Star Trek game usually uses the term *course* for a direction. The course value is a real number ranging between 1 and 9, i.e. $[1.0, 9.0]$ mathematically. Starting from 1.0 which denotes the east direction, the course value increases in anti-clockwise direction. Advancing toward 9.0. which also denotes the east direction, the course value circles around 360 degrees approaching back to the starting point. This course diagram is illustrated in Fig. A.3.

   The navigation distance is also specified in a real number. Star Trek game uses the term *Warp Factor* to represent the travelling distance. For a short-distant travelling, it takes 0.1 warp factor to travel from a sector to the next one.

Table A.1: Star Trek game's commands and arguments.

command	**argument#1**	**argument#2**	**action**	**query**
nav	direction [1.0 - 9.0]	distance (0.0 - 8.0]	navigate	-
srs	-	-	-	sector map & Enterprise report
lrs	-	-	-	neighboring galaxy map
pha	attack energy (0 - 3000]	-	phaser attack	-
tor	direction [1.0 - 9.0]	-	torpedo attack	-
she	shield energy [0 - 3000]	-	set shield	query current shield energy
dam	-	-	request for rapid repair	device damages
com	*0*	-	-	full galaxy map
	1	-	-	mission report
	2	-	-	Klingons report
	3	-	-	Starbase report
	4	initial & final coordinates	-	direction & distance
xxx	-	-	resign	-

Likewise, it takes 1.0 warp factor to travel from a quadrant to the adjacent quadrant. Navigation operates at the expense of a small amount of energy. Star Trek game blocks any attempt to travel beyond the galaxy limit. It shuts down the Enterprise at the edge sector of the galaxy. The Enterprise's warp engine can be damaged occasionally. Damage of the warp engine limits the travelling distance to 0.2 warp factor in a player's turn.

2. ***pha*** command fires a phaser attack with a specific energy. The Enterprise uses its energy to make a phaser attack. Therefore, the argument value of

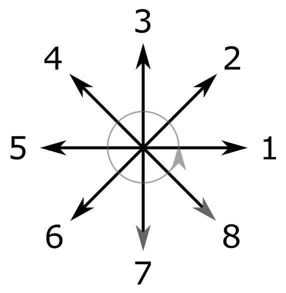


Figure A.3: Representation of course value in Star Trek game

pha command cannot exceed the Enterprise's remaining energy. Sometimes, a player lowers the Enterprise's shield to increase the phaser energy for a more powerful attack. Damage of the phaser control prevents the Enterprise from firing the phaser.

3. ***tor*** command launches a photon torpedo in a specific direction. A player specifies the torpedo direction, using the course value as demonstrated in Fig. A.3, to destroy a specific Klingon spaceship. A star along the torpedo path blocks the torpedo attack and makes the attack fail. Damage of the photon tube disables the torpedo launch. There are initially 10 torpedoes when the game starts, and the refill can always be done at a Starbase. The Enterprise cannot launch the attack when it runs out of torpedo weapons.

4. ***she*** command sets shield energy to a specific value. The Enterprise shares its energy with a defensive shield. The *she* command argument cannot exceed the remaining energy. Damage of the shield control prevents the Enterprise from adjusting the current shield energy. This adds more restrictions to the energy management.

5. ***xxx*** command quits the game in the middle of playing. A player uses this command to resign from the current game and restart a new game.

A.3.2 Query Commands

To control the Starship Enterprise, a player issues one of the following four query commands:

1. ***srs*** command shows the sector map and important status of the Enterprise. The Enterprise uses its short-range sensor (SRS) to scan all sectors within the current quadrant. This results to a sector map, as shown in the middle of Fig. A.1. The map helps a player to avoid colliding with a star in a navigation. It also helps the player to specify the effective firing direction, right toward the Klingon spaceship any without blocking stars, in a torpedo attack. When SRS is damaged, the sector map is unavailable. This greatly diminishes success of a torpedo attack due to the unknown launching direction.

   In addition to the sector map, this command also displays various important information of the Enterprise, e.g. its position in the galaxy, the current Stardate, the quadrant, the number of Klingon spaceships, shield energy, remaining energy, remaining torpedoes, etc.

2. ***lrs*** command reveals a portion of the galaxy map. The Enterprise uses its long-range sensor (LRS) to search for the number of game objects in the surrounding 3×3 quadrants centering on the Enterprise's position. The Library-Computer device records this map portion in its galaxy map storage, which is retrievable via *com* command. Damage of the LRS inhibits a quick survey over the neighboring quadrants. This requires more game time to survey the entire galaxy.

3. ***dam*** command displays the state of repair for all devices. The state of repair shows the negative value when a device is damaged. More negative value requires more time to repair, under a regular repair process. The damage control itself can also be damaged. This makes a player unable to check the repair status. Another function of *dam* command is to request for the rapid repair service when docking at the Starbase.

4. ***com*** command shows game reports and performs a support function. There are totally six options offered by the Library-Computer device of the Enterprise. Below is the list of all services.

   (a) *Cumulative Galactic Record:* shows an entire galaxy map, recorded from the LRS device.

   (b) *Status Report:* presents three major information of the game mission: the number of remaining Klingon spaceships, Stardates, and available Starbases in the mission.

   (c) *Photon Torpedo Data:* displays the direction and distance information from the Enterprise to each Klingon in the quadrant. This provides an alternative way to get the Klingon's direction for a torpedo attack.

   (d) *Starbase Nav Data:* displays the direction and distance information from the Enterprise to the Starbase in the quadrant. This provides an alternative way to navigate to the Starbase in the quadrant.

   (e) *Direction/Distance Calculator:* calculates the direction and distance between two coordinates.

(f) *Galaxy 'Region Name' Map:* presents the names of each quadrant in the galaxy.

Damage of the Library-Computer device disables all services above. In addition, it may interfere other operations of the Enterprise; for example, the torpedo path may be altered from the specified direction.

Appendix B

Simple FS Rules for Star Trek

B.1 FS Input Variables

Table B.1: A list of all FS input variables used in the simple FS rules. The table shows variable name, table name where the variable belongs to, minimum value, and maximum value of each FS variable.

	Variable Name	Table Name	Min Value	Max Value
1	SHIELD_ENERGY	SHIELD	0.0	3000.0
2	ENERGY_LEFT	NAVIGATE	0.0	3000.0
3	TIME_LEFT	NAVIGATE	0.0	100.0
4	UNFOUND_STARBASE	TO_STARBASE	0.0	5.0
5	DISTANCE_TO_STARBASE	TO_STARBASE	0.0	8.0

B.2 FS Modular Tables

Table B.2: **SHIELD** table as the root of modular tables.

	SHIELD ENERGY (2 degree)	KLINGON EXISTS (boolean)	SHIELD AVAILABLE (boolean)	STARBASE REPAIRABLE (boolean)	DECISIONS
1	LOW	YES	YES	YES	set_shield_energy()
2	LOW	YES	YES	NO	set_shield_energy()
3	LOW	YES	NO	YES	TO_STARBASE
4	LOW	YES	NO	NO	ATTACK
5	LOW	NO	YES	YES	TO_STARBASE
6	LOW	NO	YES	NO	NAVIGATE
7	LOW	NO	NO	YES	TO_STARBASE
8	LOW	NO	NO	NO	NAVIGATE
9	HIGH	YES	YES	YES	ATTACK
10	HIGH	YES	YES	NO	ATTACK
11	HIGH	YES	NO	YES	TO_STARBASE
12	HIGH	YES	O	NO	ATTACK
13	HIGH	NO	YES	YES	TO_STARBASE
14	HIGH	NO	YES	NO	NAVIGATE
15	HIGH	NO	NO	YES	TO_STARBASE
16	HIGH	NO	NO	NO	NAVIGATE

Table B.3: **ATTACK** table.

	KLINGONS HIDDEN_ALL (boolean)	TORPEDO AVAILABLE (boolean)	PHASER AVAILABLE (boolean)	DECISIONS
1	YES	YES	YES	fire_torpedo() & fire_phaser()
2	YES	YES	NO	reveal_klingon()
3	YES	NO	YES	fire_phaser()
4	YES	NO	NO	TO_STARBASE
5	NO	YES	YES	fire_torpedo() & fire_phaser()
6	NO	YES	NO	fire_torpedo()
7	NO	NO	YES	fire_phaser()
8	NO	NO	NO	TO_STARBASE

Table B.4: **NAVIGATE** table.

	WEAPON_AVAILABLE (boolean)	ENERGY_LEFT (2 degree)	TIME_LEFT (2 degree)	DECISIONS
1	YES	HIGH	HIGH	TO_KLINGON
2	YES	HIGH	LOW	TO_KLINGON
3	YES	LOW	HIGH	TO_STARBASE
4	YES	LOW	LOW	TO_KLINGON
5	NO	HIGH	HIGH	TO_STARBASE
6	NO	HIGH	LOW	TO_STARBASE
7	NO	LOW	HIGH	TO_STARBASE
8	NO	LOW	LOW	TO_STARBASE

Table B.5: **TO_STARBASE** table.

	STARBASE ON_MAP (boolean)	UNFOUND STARBASE (2 degree)	DISTANCE TO_STARBASE (2 degree)	DECISIONS
1	YES	HIGH	HIGH	navigate_to_survey()
2	YES	HIGH	LOW	navigate_to_starbase()
3	YES	LOW	HIGH	navigate_to_starbase()
4	YES	LOW	LOW	navigate_to_starbase()
5	NO	HIGH	HIGH	navigate_to_survey()
6	NO	HIGH	LOW	navigate_to_survey()
7	NO	LOW	HIGH	navigate_to_survey()
8	NO	LOW	LOW	navigate_to_survey()

Table B.6: **TO_KLINGON** table.

	KLINGON_ON_MAP (boolean)	DECISIONS
1	YES	navigate_to_klingons()
2	NO	navigate_to_survey()

Appendix C

Extended FS Rules for Star Trek

C.1 FS Input Variables

Table C.1: A list of all FS variables used in the extended FS rules. The table shows variable name, table name where the variable belongs to, minimum value, and maximum value of each FS variable. The extended rules consist of 14 FS input variables in the first half and 6 FS output variables in the last half of the table.

	Variable Name	Table Name	Min Value	Max Value
1	DISTANCE_TO_STARBASE	NAVIGATE	0.0	8.0
2	DAMAGING_DEVICES	NAVIGATE	0.0	10.0
3	ENERGY_LEFT	NAVIGATE	0.0	3000.0
4	TIME_LEFT	NAVIGATE	0.0	60.0
5	N_KLINGONS_IN_QUADRANT	ATTACK	0.0	3.0
6	DISTANCE_TO_KLINGON	TO_KLINGON	0.0	8.0
7	TIME_LEFT	TO_KLINGON	0.0	60.0
8	UNFOUND_STARBASE	TO_STARBASE	0.0	5.0
9	DISTANCE_TO_STARBASE	TO_STARBASE	0.0	8.0
10	SHIELD_ENERGY	RED_ALERT	0.0	3000.0
11	N_KLINGONS_IN_QUADRANT	$SHIELD	0.0	3.0
12	DISTANCE_TO_KLINGON	$SHIELD	0.0	8.0
13	N_KLINGONS_IN_QUADRANT	$PHASER	0.0	3.0
14	DISTANCE_TO_KLINGON	$PHASER	0.0	8.0
15	$SHIELD_LOW	$SHIELD	0.0	3000.0
16	$SHIELD_MEDIUM	$SHIELD	0.0	3000.0
17	$SHIELD_HIGH	$SHIELD	0.0	3000.0
18	$PHASER_LOW	$PHASER	0.0	3000.0
19	$PHASER_MEDIUM	$PHASER	0.0	3000.0
20	$PHASER_HIGH	$PHASER	0.0	3000.0

C.2 FS Decision Tables

Table C.2: **RED_ALERT** table as a root of the critical warning.

	SHIELD AVAILABLE (boolean)	STARBASE REPAIRABLE (boolean)	SHIELD ENERGY (3 degree)	DECISIONS
1	NO	NO	LOW	ATTACK
2	NO	NO	MEDIUM	ATTACK
3	NO	NO	HIGH	ATTACK
4	NO	YES	LOW	ATTACK
5	NO	YES	MEDIUM	ATTACK & TO_STARBASE
6	NO	YES	HIGH	ATTACK & TO_STARBASE
7	YES	NO	LOW	set_shield_energy()
8	YES	NO	MEDIUM	set_shield_energy() & ATTACK
9	YES	NO	HIGH	ATTACK
10	YES	YES	LOW	set_shield_energy()
11	YES	YES	MEDIUM	set_shield_energy() & ATTACK
12	YES	YES	HIGH	ATTACK

Table C.3: **MAIN_DECISION** table as a root of modular tables.

	KLINGON_IN QUADRANT (boolean)	STARBASE REPAIRABLE (boolean)	DECISIONS
1	NO	NO	NAVIGATE
2	NO	YES	TO_STARBASE
3	YES	NO	ATTACK
4	YES	YES	ATTACK

Table C.4: **ATTACK** table.

	N_KLINGONS IN_QUADRANT (2 degree)	KLINGONS HIDDEN_ALL (boolean)	TORPEDO AVAILABLE (boolean)	PHASER AVAILABLE (boolean)	DECISIONS
1	LOW	NO	NO	NO	TO_STARBASE
2	LOW	NO	NO	YES	fire_phaser()
3	LOW	NO	YES	NO	fire_torpedo()
4	LOW	NO	YES	YES	fire_torpedo() & fire_phaser()
5	LOW	YES	NO	NO	TO_STARBASE
6	LOW	YES	NO	YES	fire_phaser()
7	LOW	YES	YES	NO	reveal_klingon()
8	LOW	YES	YES	YES	reveal_klingon() & fire_phaser()
9	HIGH	NO	NO	NO	TO_STARBASE
10	HIGH	NO	NO	YES	fire_phaser()
11	HIGH	NO	YES	NO	fire_torpedo()
12	HIGH	NO	YES	YES	fire_phaser()
13	HIGH	YES	NO	NO	TO_STARBASE
14	HIGH	YES	NO	YES	fire_phaser()
15	HIGH	YES	YES	NO	reveal_klingon()
16	HIGH	YES	YES	YES	fire_phaser()

Table C.5: **NAVIGATE** table.

	DISTANCE_TO STARBASE (2 degree)	DAMAGING DEVICES (2 degree)	ENERGY LEFT (2 degree)	TIME LEFT (2 degree)	DECISIONS
1	LOW	LOW	LOW	LOW	TO_KLINGON
2	LOW	LOW	LOW	HIGH	TO_STARBASE
3	LOW	LOW	HIGH	LOW	TO_KLINGON
4	LOW	LOW	HIGH	HIGH	TO_STARBASE
5	LOW	HIGH	LOW	LOW	TO_STARBASE
6	LOW	HIGH	LOW	HIGH	TO_STARBASE
7	LOW	HIGH	HIGH	LOW	TO_KLINGON
8	LOW	HIGH	HIGH	HIGH	TO_STARBASE
9	HIGH	LOW	LOW	LOW	TO_KLINGON
10	HIGH	LOW	LOW	HIGH	TO_KLINGON
11	HIGH	LOW	HIGH	LOW	TO_KLINGON
12	HIGH	LOW	HIGH	HIGH	TO_KLINGON
13	HIGH	HIGH	LOW	LOW	TO_KLINGON
14	HIGH	HIGH	LOW	HIGH	TO_STARBASE
15	HIGH	HIGH	HIGH	LOW	TO_KLINGON
16	HIGH	HIGH	HIGH	HIGH	TO_STARBASE

Table C.6: **TO_STARBASE** table.

	STARBASE ON_MAP (boolean)	UNFOUND STARBASE (2 degree)	DISTANCE_TO STARBASE (2 degree)	DECISIONS
1	NO	LOW	LOW	navigate_to_survey()
2	NO	LOW	HIGH	navigate_to_survey()
3	NO	HIGH	LOW	navigate_to_survey()
4	NO	HIGH	HIGH	navigate_to_survey()
5	YES	LOW	LOW	navigate_to_starbase()
6	YES	LOW	HIGH	navigate_to_starbase()
7	YES	HIGH	LOW	navigate_to_starbase()
8	YES	HIGH	HIGH	navigate_to_survey()

Table C.7: **TO_KLINGON** table.

	DISTANCE_TO_KLINGON (3 degree)	TIME_LEFT (2 degree)	DECISIONS
1	LOW	LOW	navigate_to_klingons()
2	LOW	HIGH	navigate_to_klingons()
3	MEDIUM	LOW	navigate_to_klingons()
4	MEDIUM	HIGH	navigate_to_survey()
5	HIGH	LOW	navigate_to_klingons()
6	HIGH	HIGH	navigate_to_survey()

Table C.8: **VALUE_SHIELD** table for command argument.

	N_KLINGONS IN_QUADRANT (2 degree)	DISTANCE TO_KLINGON (2 degree)	VALUES
1	LOW	LOW	$SHIELD_MEDIUM
2	LOW	HIGH	$SHIELD_LOW
3	HIGH	LOW	$SHIELD_HIGH
4	HIGH	HIGH	$SHIELD_MEDIUM

Table C.9: **VALUE_PHASER** table for command argument.

	N_KLINGONS IN_QUADRANT (2 degree)	DISTANCE TO_KLINGON (2 degree)	VALUES
1	LOW	LOW	$PHASER_LOW
2	LOW	HIGH	$PHASER_MEDIUM
3	HIGH	LOW	$PHASER_MEDIUM
4	HIGH	HIGH	$PHASER_HIGH