Road to an Interesting Opponent: An Agent that Predicts the Users' Combination Attacks in a Fighting Videogame

Simon E. Ortiz B.[1*], Koichi Moriyama[2], Mitsuhiro Matsumoto[1],
Ken-ichi Fukui[2], Satoshi Kurihara[2], and Masayuki Numao[2]

[1]Graduate School of Information Science and Technology, Osaka University
[2]Institute of Scientific and Industrial Research, Osaka University

Abstract: In fighting videogames users usually prefer playing against other users rather than against the machine. We are assuming that the adaptability of the human player makes it interesting. We are aiming to produce an agent for a fighting videogame that can adapt to its users, allowing users to enjoy the game even when playing alone. We have completed a part of this continuing project: an agent that predicts the users' combination attacks. We introduce this agent and present the results of the experiments.

1 Introduction

Fighting videogames are a popular genre of videogames with new titles being released every year. A fighting videogame is basically a simulation of hand-to-hand combat. Fights are carried out in a manner similar to boxing matches: usually there are two participants, there are several rounds with a given time limit, etc. The winner of each round is the user that lowers the energy of the opponent to zero by means of attacks.

All fighting videogames are designed to be played by at least two users competitively, i.e. users playing against one another. However, these games also have the possibility of being played by only one user. In this case, the machine will take control of the opponent. Users generally choose to play against the machine in order to advance the story of the game, to practice, or because there is nobody around to play with. But, if given the option, users usually prefer to play against other users.

One of the reasons users prefer to play against other users could be that the AI that controls the character in solo mode is usually uninteresting. This is not to say that it is easy to defeat, since the machine can execute complicated attacks and respond quickly. Rather, we are assuming that the adaptability of the human player makes it interesting compared to the machine. However, the machine AI used so far is typically of a simple design, just enough to make the player feel the software is reasonably smart [1], which means that standard videogames' AI is not complex enough to learn users' patterns.

When two users play against each other, they usually fight a battle beyond quick button mashing. Each user has a strategy they follow and patterns they develop. The fun part of the game is trying to learn each other's technique and predict the future actions of his/her opponent. We are aiming to produce an agent for a fighting videogame that can adapt to its users, simulating the learning of each other's technique. We hope this will allow the users to enjoy the game even when playing alone.

In this paper, we present the model of our adapting agent for a fighting videogame. Afterwards, we present the experiments and results for one of its modules: the Receiving-Combo Sub-agent.

2 The Fighting Videogame

Typical fighting videogames are one-on-one games. In solo mode the user is in control of one character, and the AI controls the opponent.

Each character starts with a predefined amount of HP (health points), when their HP reaches zero they lose. The objective is to defeat the other character first. Usually the fight is decided to the best of several rounds, each round has a defined time limit.

Fighting games have several stages where the fights occur. The characters can walk back and forward, jump and crouch in a 2D plane of the stage. If the stage has edges, falling from the edge typically means immediately losing the round.

Characters have normal attack actions, for example *punch*, *kick*, and projectile-like long range *special* attacks

*Contact: ISIR, Osaka University
Address: 8-1 Mihogaoka, Ibaraki, Osaka, 567-0047, Japan
e-mail: ortiz@ai.sanken.osaka-u.ac.jp

(from now on we will abbreviate these actions as `p`, `k` and `s` respectively). Normal attacks deal low amounts of damage compared to combos (combos will be explained in detail later). There is usually a *block* action. If a character is using *block*, normal attacks from the opponent will have no effect. The *block* action deals no damage.

2.1 Combos

Combos, short for combination attacks, are a common game design element found in most modern fighting videogames. A combo deals greater damage than normal attacks.

Different games have a different approach to combos. We deal with combos of the form of predefined sequences of normal attacks successfully executed within a brief time limit. For instance, let us consider the combos shown in Table 1. The sequence of four *punches* in a row `pppp` is a combo. When a character successfully connects four *punches* the opponent will receive the normal amount of damage of each normal attack plus the extra amount of damage defined for this combo. If the first two actions of a combo are connected then the rest of the actions of the combo cannot be blocked using the *block* action.

Table 1: Combos.

ID	Actions	Damage	ID	Actions	Damage
0	pppp	15	6	kppk	20
1	pppk	20	7	kpps	25
2	ppps	25	8	kspp	15
3	pkpp	20	9	ksps	20
4	pkpk	51	10	kpsp	30
5	pkps	25	11	kkkk	30

2.2 Combo-breakers

Combo-breakers are a way of blocking or counter-attacking a combo.

For some games, including the case we deal with, combo-breakers are the only ways to defend from a combo. The combo-breaker is one predefined action or sequence of actions that can be executed by the character receiving the combo. In order to break the combo (i.e. to counter-attack it) the action should be executed before the combo is completed. When a combo is broken, the character receiving the combo will not receive the extra damage, instead the character executing the combo will receive it.

An example will clarify this concept. Let us use again the combos in Table 1. The sequence `pppp` is defined as a combo. Let us assume that this combo in particular has the action `p` defined as its combo-breaker. If character A is executing the combo `pppp`, and character B executed `p` before A's fourth `p`, then the combo will be broken and A will receive the extra damage. Instead, if B didn't execute `p` before A's fourth `p`, or if B's last action was not `p`, the combo will not be broken and B will receive the extra damage of the combo.

3 Agent as an Interesting Opponent

Our goal is to produce an agent for fighting videogames that can adapt to the users fighting style, allowing users to enjoy the game even when playing alone.

Playing a fighting game can be thought of as dealing with three tasks: executing normal attacks plus moving, executing combos, and executing combo-breakers when receiving a combo.

The agent is divided into three sub-agents: Main Sub-agent, Executing-Combo Sub-agent and Receiving-Combo Sub-agent. Dividing tasks should help the agent learn each task in less time.

The Main Sub-agent will move the character and execute normal attacks. When deemed appropriate it will pass the control to one of the other sub-agents. The Executing-Combo Sub-Agent will execute combos of a difficulty similar to those of the user's by generating a set of combos with similar features to those of the set of the user’s combos. The Receiving-Combo Sub-agent will learn the combo patterns of the user and try to execute the appropriate combo-breakers.

The agent composed of the mentioned sub-agents will be able to move away from the user or get close to him/her depending on whether the user will attack or not, execute combos appropriate to the level of the user, and learn the user’s combo patterns to response adequately.

We have completed a part of this continuing project: the Receiving-Combo Sub-agent (referred to henceforth as RCSA). Next we introduce the model behind the RCSA.

3.1 Receiving-Combo Sub-agent

The objective of the RCSA is to choose the proper combo breaker for the combos executed by the user.

Since the combo-breaker must be executed before the last action of a combo, the problem becomes predicting the full combo given its beginning and the combos

previously executed by the user.

Since videogames must run in real-time, all the learning is delayed until the end of each episode. The resulting data is stored in a way that allows for quick decisions in real-time when used in the next episode.

We propose utilizing Substring Tree [2], a pattern mining technique, to learn the user's pattern and predict the adequate combo-breaker.

3.2 Pattern learning for Combo-breaking

Although studies have yet to be conducted, it is reasonable to expect that users develop patterns and routines in fighting videogames, because these patterns allow the user to execute a series of attacks quickly without having to decide each attack and a series of quick attacks is harder to defend from.

For this agent we utilize Substring Tree [2], an algorithm for mining frequent spatio-temporal data. Adapting the algorithm to mine combo patterns is trivial: the discrete spatial regions originally defined for the algorithm are substituted by the IDs of the combos. Then the algorithm is used as presented in [2].

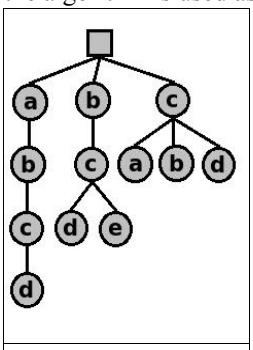

Figure 1: Combo Tree

The Substring Tree algorithm returns the set of frequent sequences of combos used by the user and their frequency. We transform this data into a decision tree where the nodes contain combos with their frequency. We start with the empty tree, and insert each frequent sequence of combos beneath the empty root node. Sequences with the same beginning will share parts of the tree. For example, if Substring Tree returns {{a, b, c, d}, {b, c, d}, {b, c, e}, {c, a}, {c, b}, {c, d}} then the decision tree will be the one shown in Figure 1.

Since the user can have patterns of various lengths, the agent tracks all possible patterns up to length *n* simultaneously. In order to do so, the agent traverses the tree following the combos recently executed by the user during the episode. The agent uses *n* pointers to the tree at the same time. Each of the *n* pointers tracks a sequence of size 1, 2, …, *n* combos. For example, if the user executed combos *a*, *b*, *c* then the first pointer will be at the rightmost branch of Figure 1, the branch that starts with *c*; the second pointer will point at node *c* from the center branch, below *b*; the third pointer would point to node *c* from the leftmost branch, below *b* below *a*; etc. If for a given pointer that tracks patterns of length *l*, the sequence of combos executed by the user does not match a sequence of combos of length *l* in the tree, then the corresponding pointer will be considered invalid.

```
action decide_combo_breaker():
   action combo_head[m] := the first
          m actions of the combo being
          executed
  combo useq[n] := last n combos executed
          by the user, most recent first
  pointer p[n]
  roulette := probabilistic roulette
  for 1 <= i <= n :
     p[i] := track(useq, i)
  for all pointers p != null :
     for all children c of p :
        if c.head == combo_head :
           roulette.add(c, c.frequency)
  return roulette.random()

node track (combo seq[], int q):
  pointer node := decision_tree.root
  for 1 <= i <= q :
     if node.has_child(seq[i]) :
        node := node.get_child(seq[i])
     else :
        node := null
        break
  return node
```

Figure 2: *Pattern Learning*'s combo-breaker selection algorithm.

The combo-breaker is decided following the algorithm in Figure 2. What the agent basically does is this: for all the pointers pointing to a valid node, it searches beneath them for a partial match for the combo being executed at the current time. It chooses a combo-breaker stochastically by their relative frequency among the breakers found under the pointers.

4 Evaluation

In order to carry out our research we created a simple fighting videogame using C++ and the open source real-time 3D graphics engine Crystal Space [3]. As can be seen in Figure 3 the game is very simplistic. The characters have the normal attacks `p`, `k` and `s` (the projectile is a sphere that moves linearly for 3 seconds before disappearing), and the *block* action. The initial HP is 200. Each fight is composed of only one round without

time limit. From the agent's perspective, one round is one episode.

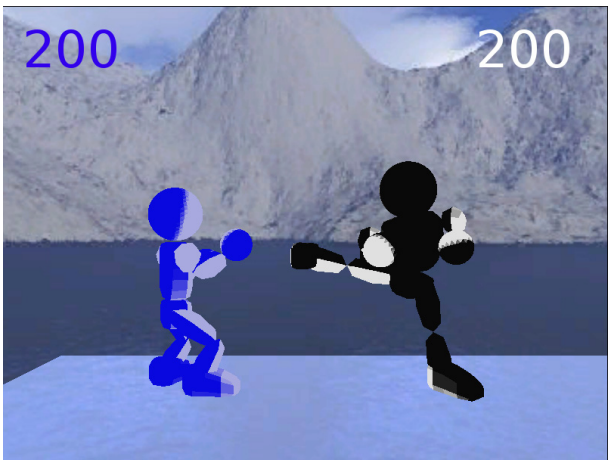

Figure 3: A screenshot of our simplistic fighting game.

The combos are defined as fixed sequences of normal attacks successfully executed within 0.5 seconds between each attack. The combos used are those shown in Table 1. The combo-breakers are defined for each combo as the last action of the combo. If the character receiving the combo executed more than one action, only the last action will be considered as the combo-breaker candidate.

For the *Pattern Learning* agent we defined the length of combos to be tracked to be five, and the size of the *head* of the combos (the parameter *m* of the algorithm `decide_combo_breaker` in Figure 2) to be three.

For comparison purposes we defined three more agents: the random agent, *Bandit* and *Time-weighted Bandit*. The reason behind our selection of these methods is that they are AI that is simple enough to be implemented in a standard fighting videogame, yet they allow the agent to adapt to the user to a certain degree. If we would like to compare our proposed agent with standard game agents these could be a good approximation.

4.1 Bandit

This agent models the problem of choosing a combo-breaker in a manner similar to the *n*-armed Bandit Problem [4]. The *n*-armed Bandit Problem is the problem of deciding which of *n* available actions execute in one unique state. Each action gives an immediate reward. The unique state is presented several times to the agent. The goal of the agent is to maximize the outcome in a defined number of episodes.

In our case, the unique state is defined by the *head* of a combo. The available actions are the normal attacks `p`, `k`, `s`. The learning method is calculating the proportion of each last action chosen by the user in a given state. The action is chosen by a random roulette, where each action has a probability of being chosen equal to its proportion of appearance.

4.2 Time-weighted bandit

This learning method is similar to *Bandit* with the difference that the weight of each action for a given state does not depends only on the number of times it appears, but on how recently it was used.

For a given state, the weight *w* of a given action *a* will be updated as follows: in the case in which action *a* was chosen by the user at time *t*

$$w_n(a) = \frac{\alpha w_{n-1}(a) + 1}{\sum_{i=0}^{n} \alpha^i}$$

for all the other actions *z* not chosen by the user at time *t*

$$w_n(z) = \frac{\alpha w_{n-1}(z)}{\sum_{i=0}^{n} \alpha^i}$$

where α is a discount factor equal to 0.9.

This method for calculating the weights is based on the formula for *eligibility traces* used in the algorithm TD(λ) from the Reinforcement Learning field [4]. Basically, we are incrementing the weight of the action used more recently and discounting all the other available actions. This value is normalized by the summation of the discount factors, which allows the selection of the action to be probabilistic, similar to the *Bandit* method.

4.3 Experiments

For each experiment we conducted four trials. After each trial the learning process was reset.

The tests were conducted through a simulated user. This simulated user obeys the following four patterns to execute combos: pattern 1: {5, 3, 8, 10, 2, 4}; pattern 2: {0, 2, 9, 1, 7, 2}; pattern 3: {5, 11, 2, 10, 3}; pattern 4: {1, 8, 11, 9}.

Each pattern was chosen using a random number generator. The number inside the patterns represents the ID of a combo in Table 1. To test the learning abilities of the agents for different pattern sizes we chose patterns of size 6, 6, 5 and 4 respectively.

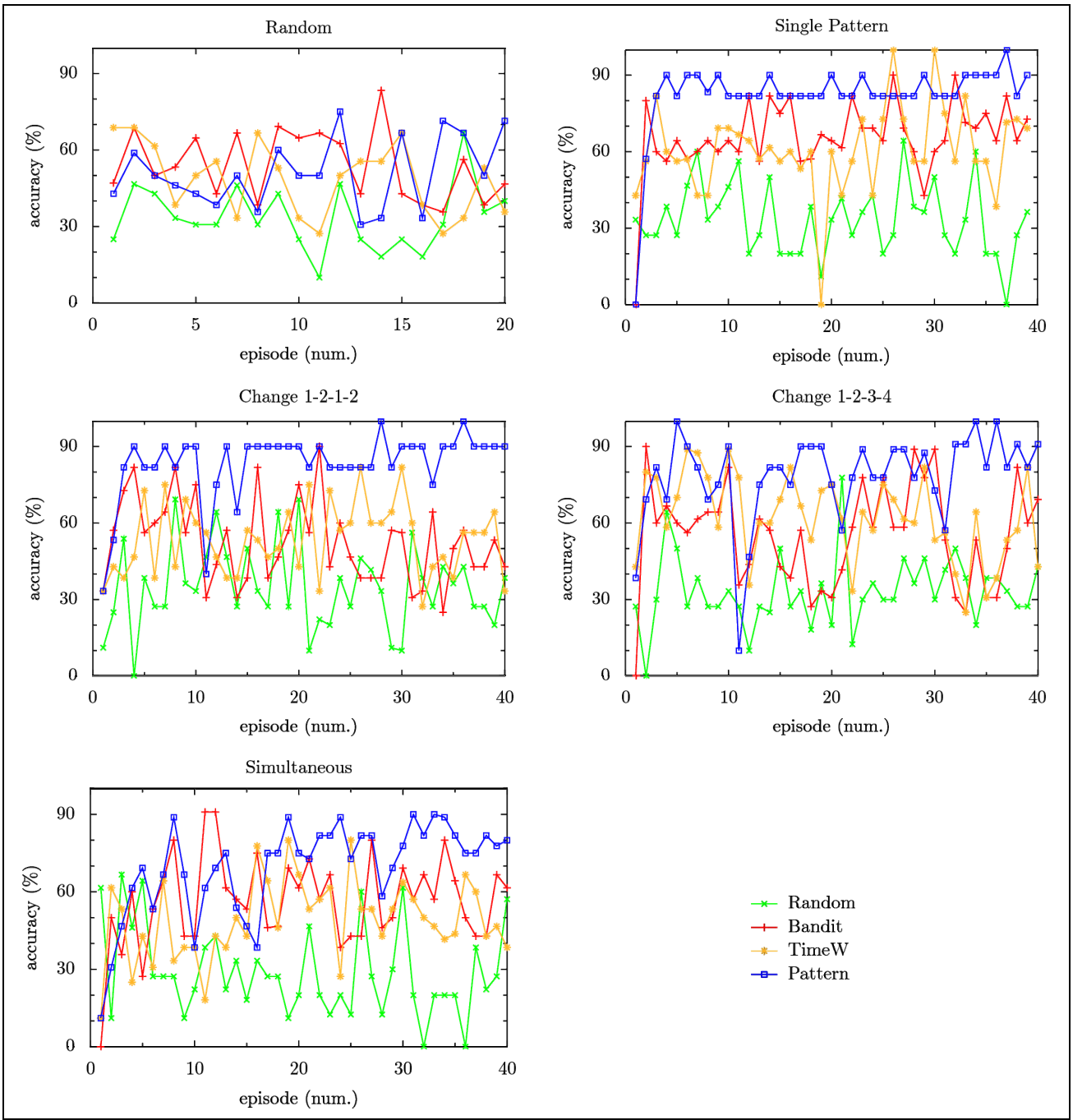

Figure 4: Results from the experiments.

We ran five different simulations:

-Random: The combos executed by the simulated user were chosen randomly, not following any pattern.

-Single: The simulated user executes combos following pattern 1.

-Change 1-2-1-2: For the first 10 episodes the simulated user executes pattern 1, for the next 10 episodes use pattern 2, repeat once.

-Change 1-2-3-4: The simulated user executes patterns 1, 2, 3 and 4 for 10 episodes each.

-Simultaneous: The simulated user chooses a pattern randomly, executes the combos defined in it, then chooses a new pattern randomly.

We measured the accuracy of each method for each episode. The accuracy is defined as the number of successfully executed combo-breakers divided by the number of combos presented to the agent. The results are shown in Figure 4. For brevity, we chose to show a representative result from each simulation.

As expected, the learning from the *Random* simulation does not seem to converge to any value. There is no information to be learned.

From the *Single* simulation we see that *Pattern Learning* can learn one single pattern in few episodes, usually around three. This agent improves the accuracy of its predictions very fast and seems to converge to around 85% accuracy. As expected, the random agent has an average accuracy of 33% (there are three possible actions out of which there's only one valid). Both *Bandit* and *Time-weighted Bandit* learn combo-breakers, but are dominated by *Pattern Learning*.

From *Change 1-2-1-2* we can see that *Pattern Learning* gets confused when presented for the first time with a new pattern, but it adapts in a couple of episodes. When presented with previously learned patterns there seems to be no confusion. Both *Bandits* also get confused when presented with a new pattern, and also get confused when presented with previous patterns. This is due to the fact that the most recent patterns have a different combo-breaker distribution from the learned patterns and these methods cannot distinguish between distributions.

In *Change 1-2-3-4* the *Pattern Learning* agent quickly recovers as in *Change 1-2-1-2*. The *Time-weighted Bandit* also got confused as in previous simulations. *Bandit* improved its accuracy from pattern 2 to pattern 3. This is possibly due to the fact that in pattern 3, combos 10 and 11 only have one possible breaker and combo 2 is presented isolated from combos with similar *heads*.

In the case of the *Simultaneous* simulation, it can be seen that *Pattern Learning* can manage more than one pattern at a time. However, it takes nearly five times more episodes to get close to the previous 85% accuracy. This can be due to the fact that it takes several episodes to experience all the available patterns. Considering the extra damage assigned to each combo and the HP of 200, each episode finishes after the execution of around 10 combos. This means that each episode the agent only experiences two patterns, the second one possible incompletely. The *Bandits* are dominated by *Pattern Learning* most of the time, and their behavior seems less accurate. The different distribution of the combo-breakers for each pattern affects their accuracy prediction.

5 Related works

Academic level AI has been applied to learning how to fight in a fighting videogame without considering adapting to the user [5]. Reinforcement learning has been applied to adapting the opponents of a videogame to its users, in this case it was not a fighting videogame, it was Pac-Man [6].

6 Conclusions

Pattern Learning is the method better suited to learning to execute combo-breakers because it can identify the context of each combo and it tends to an accuracy of 85%. The reason why it does not have an accuracy of 100% even in the *Single* simulation is due to the fact that it is tracking more than one possible pattern and chooses the combo-breaker stochastically based on its frequency.

Is the *Pattern Mining* RCSA a more interesting candidate for our fighting videogame? At this stage it is hard to know. In the final version of the game, predicting the combos of the user will be just one part of the entire system. We have not found in the literature which is the most interesting win-lose rate for a fighting videogame. If a more difficult opponent is a more interesting opponent, then the *Pattern Learning* agent would be our best choice.

References

[1] Adams E.: Fundamentals of Game Design (2nd ed.), New Riders (2009)

[2] Cao H., Mamoulis N., and Cheung D.: Mining Frequent Spatio-temporal Sequential Patterns, *Proceedings of the Fifth IEEE International Conference on Data Mining*, pp. 82-89 (2005).

[3] http://www.crystalspace3d.org/

[4] Sutton R., Barto A.: Reinforcement Learning, An Introduction, The MIT Press (1999)

[5] Graepel T., Herbrich R., Gold J.: Learning to Fight, *Proceedings of Computer Games: Artificial Intelligence, Design and Education*, pp. 193-200 (2004)

[6] Yannakakis G., Hallam J.: Evolving Opponents for Interesting Interactive Computer Games, *Proceedings of the Eighth International Conference on the Simulation of Adaptive Behavior; From Animals to Animats 8*, pp. 499-508 (2004)