Challenge-Sensitive Action Selection: an Application to Game Balancing

Gustavo Andrade
Geber Ramalho
Hugo Santana
Universidade Federal de Pernambuco
Centro de Informática
Caixa Postal 7851, CEP 50732-970, Recife, Brazil
{gda,glr,hps}@cin.ufpe.br

Vincent Corruble
Université Paris 6
Laboratoire d'Informatique de Paris VI
Bôite 169 - 4 Place Jussieu
75252 PARIS CEDEX 05
Vincent.Corruble@lip6.fr

Abstract

Dealing with users of different skills, and of variable capacity for learning and adapting over time, is a key issue in Human-Machine Interaction, particularly in highly interactive applications such as computer games. Indeed, a recognized major concern for the game developers' community is to provide mechanisms to dynamically balance the difficulty level of the games in order to keep the user interested in playing. This work presents an innovative use of reinforcement learning techniques to build intelligent agents that adapt their behavior in order to provide dynamic game balancing. The idea is to couple learning with an action selection mechanism which depends on the evaluation of the current user's skills. To validate our approach, we applied it to a real-time fighting game, obtaining good results, as the adaptive agent is able to quickly play at the same level as opponents with different skills.

1. Introduction

Considering the individual characteristics of users, as well as their capacity to learn and adapt, is a key issue when trying to provide high usability to computer systems [12]. This is a hard problem in applications where there is a great diversity among users in terms of skills and/or domain knowledge. Moreover, users may improve their performance at different rates and use different learning strategies [12]. It is then necessary to provide user adaptation mechanisms when building interactive computer systems [9]. This need for user adaptation is particularly a major concern in applications such as computer aided instruction [3] and computer games. In these cases, the skill or knowledge of the user must be continuously assessed to guide the choice of the adequate challenges (e.g., missions, exercises, questions, etc.) to be proposed.

In computer games, one of the most interactive domains nowadays, the issue of providing a good level of challenge for the user is referred to as *game balancing*, and is recognized by the game development community as a key characteristic for a successful game [5]. Balancing a game consists in changing parameters, scenarios and behaviors in order to avoid the extremes of getting the player frustrated because the game is too hard or becoming bored because the game is too easy [8]. The idea is to keep the user interested in playing the game from the beginning to the end. Unfortunately, fixing a few pre-defined and static difficulty levels (e.g., beginner, intermediate and advanced) is not sufficient. In fact, not only should the classification of users' skill levels be fine-grained, but the game difficulty should also follow the players' personal evolutions, as they make progress via learning, or as they regress (for instance, after a long period without playing the game).

The current approaches to providing dynamic game balancing are either tightly dependent on the game style, or do not present yet adequate mechanisms to ensure a fast and close adaptation to the diversity of users' skill levels, as well as their evolution.

In order to deal with the problem, we avoided the drawbacks of attempting to learn to play at the same level as the user. Instead, we coupled two mechanisms. First, we build agents that are capable of learning optimal strategies, using Reinforcement Learning (RL). These agents are trained offline to exhibit a good initial performance level and keep learning during the game. Second, we provide adaptation to agents via an original action selection mechanism dependent on the difficulty the human player is currently facing. We validated our approach empirically, applying it to a real-time two-opponent fighting game called Knock'Em [2], whose functionalities are similar to those of successful commercial games, such as Capcom Street Fighter and Midway Mortal Kombat.

In the next section we introduce the problems in providing dynamic game balancing. Then, we shortly present some RL concepts and discuss its application to games. In Section 4 and 5, respectively, we introduce our approach and show its application to a specific game. Finally, we present some conclusions and ongoing work.

2. Dynamic game balancing approaches

Dynamic game balancing is a process which must satisfy at least three basic requirements. First, the game must, as quickly as possible, identify and adapt itself to the human player's initial level, which can vary widely from novices to experts. Second, the game must track as close and as fast as possible the evolutions and regressions in the player's performance. Third, in adapting itself, the behavior of the game must remain believable, since the user is not meant to perceive that the computer is somehow facilitating things (e.g., by decreasing the agents physical attributes or executing clearly random and inefficient actions).

There are many different approaches to address dynamic game balancing. In all cases, it is necessary to measure, implicitly or explicitly, the difficulty the user is facing at a given moment. This measure can be performed by a heuristic function, which some authors [4] call a "challenge function". This function is supposed to map a given game state into a value that specifies how easy or difficult the game feels to the user at that specific moment. Examples of heuristics used are: the rate of successful shots or hits, the numbers of won and lost pieces, life point evolution, time to complete a task, or any metric used to calculate a game score.

Hunicke and Chapman's approach [6] controls the game environment settings in order to make challenges easier or harder. For example, if the game is too hard, the player gets more weapons, recovers life points faster or faces fewer opponents. Although this approach may be effective, its application is constrained to game genres where such environment manipulations are possible.

Another approach to dynamic game balancing is to modify the behavior of the Non-Player Characters (NPCs), characters controlled by the computer and usually modeled as intelligent agents. A traditional implementation of such an agent's intelligence is to use behavior rules, defined during game development. A typical rule in a fighting game would state "*punch opponent if he is reachable, chase him, otherwise*". Extending such an approach to include opponent modeling can be made through dynamic scripting [15], which assigns to each rule a probability of being picked. Rule weights are dynamically updated throughout the game. The use of this technique to game balancing can be made by not choosing the best rule, but the closest one to the user level. However, as game complexity increases, this technique results in a lot of rules, which are error-prone and hard to build and maintain. Moreover, the agent performance becomes limited by the best designed rule, which can not be good enough for very skilled users.

A natural approach to address the problem is to use machine learning [9]. Demasi and Cruz [4] built intelligent agents employing genetic algorithm techniques to keep alive those agents that best fit the user level. Online coevolution [19] is used in order to speed up the learning process. Online coevolution uses pre-defined models (agents with good genetic features) as parents in the genetic operations, so that the evolution is biased by them. These models are constructed by offline training or by hand, when the agent genetic encoding is simple enough. This is an innovative approach, indeed. However, it shows some limitations when considering the requirements stated before. Using pre-defined models, the agent's learning becomes restricted by these models, jeopardizing the application of the technique for very skilled users or users with uncommon behavior. As these users do not have a model to speed up learning, it takes a long time until the agents reaches the user level. Furthermore, this approach works only to increase the agent's performance level. If the player's skills regress, the agents cannot regress together. This limitation compels the agent to always start the evolution from the easiest level. While this can be a good strategy when the player is a beginner, it can be bothering for skilled players, since they will probably need to wait a lot for the evolution of the agents.

3. Reinforcement learning in games

3.1. Background

Reinforcement Learning (RL) is often characterized as the problem of "learning what to do (how to map situations into actions) so as to maximize a numerical reward signal" [16]. This framework is often defined in terms of the Markov Decision Processes (MDP) formalism, in which we have an agent that sequentially makes decisions in an environment: it perceives the current state s, from a finite set S, chooses an action a, from a finite set A, reaches a new state s' and receives an immediate reward signal $r(s,a)$. The information encoded in s should satisfy the Markov Property, that is, it should summarize all present and past sensations in such a way that all relevant information is retained. Implicitly, the reward signal *r(s,a)* determines the agent's objective: it is the feedback which guides the desired behavior.

The main goal is to maximize a long-term performance criterion, called return, which represents the expected value of future rewards. The agent then tries to learn an *optimal policy* π^* which maximizes the expected return. A policy is a function $\pi(s) \rightarrow a$ that maps state perceptions into actions. Another concept in this formalism is the *action-value function*, $Q^{\pi}(s,a)$, defined as the expected return when starting from state *s*, performing action *a*, and then following π thereafter. If the agent can learn the optimal action-value function $Q^*(s,a)$, an optimal policy can be constructed greedily: for each state *s*, the best action *a* is the one that maximizes $Q^*(s,a)$.

A traditional algorithm for solving MDPs is Q-Learning [18]. It consists on iteratively computing the values for the action-value function, using the following update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma.V(s') - Q(s,a)]$$

in which $V(s') = \max_a Q(s',a)$, α is the learning rate and γ is a discount factor, which represents the relative importance of future against immediate rewards.

There are some particular characteristics of RL which makes it attractive to complex applications like computer games. First, different from other kinds of machine learning techniques, it does not require any previous example of good behavior, being able to find optimal strategies only through trial and error, thus reducing the development effort necessary to build the AI of the game. Furthermore, it can be applied offline, as a pre-processing step during the development of the game, and then be continuously improved online after its release [11].

RL has been successfully used in board games, like backgammon [17], checkers [14] and Go [1]. This technique has also been applied successfully in other domains such as robotic soccer [13]. The work here presented differs from these mainly in two different aspects. First, many of these applications are turn-based games, in which the environment does not change while the agent is choosing its action. We deal in this work with real-time, dynamic games, which lead in general to more complex state representations, and the need to address time processing issues. Second, while these works are basically interested in making the computer beat any opponent (possibly optimal), our goal is to have the computer always adequately challenge his opponent, whether or not he/she is playing optimally.

3.2. Learning to play directly at the user level

The problem of dynamically changing the game level could be addressed with RL by carefully choosing the reward so that the agent learns to act in the same level of the user skill. When the game is too easy or too hard a negative reward is given to the agent, otherwise the feedback is a positive reward.

This approach has the clear benefit that the mathematical model of learning actually corresponds to the goals of the agent. However, this approach has a lot of disadvantages, with respect to the requirements stated in Section 2. First, using this approach, the agent will not be able immediately to fight against expert players, since it would need to learn before. Second, this learning approach may lead to non-believable behaviors. For instance, in a fight game such as Knock'em, the agent can learn that after hitting the user hard, it must be static and use no defense, letting the user hit him back, so as the overall game score remain balanced.

4. Challenge-sensitive action selection

Given the difficulties in directly learning to play at the user level, an original alternative is to face dynamic game balancing as two separate problems: learning (building agents that can learn optimal strategies) and adapting (providing action selection mechanisms for providing game balance, possibly using sub-optimal actions).

Our approach faces the first problem with reinforcement learning. Due to the requirement of being immediately able to play at the human player level, including expert ones, at the beginning of the game, offline training is needed to bootstrap the learning process. This can be done by letting the agent play against itself (self-learning) [7] or other pre-programmed agents [10]. Then, online learning is used to adapt continually this initially built-in intelligence to the specific human opponent, in order to discover the most adapted strategy to play against him or her.

Concerning adaptation, the idea is to find the adequate policy for choosing actions that provide a good game balance, i.e., actions that keep agent and human player at approximately the same performance level. Once the agent has learned the optimal policy, it could at times (with a frequency depending on the challenge function, i.e. on the difficulty the human player is facing) simply choose randomly the action to be executed. The problem with this approach is that it can easily violate the believability requirement stated in Section 2. For instance, in the Knock'em case, one of the possible actions is "*jump backwards*", which is a sensible action when being attacked, but an extremely weird one when there is no apparent danger. Another example is to punch even when the adversary is very far.

In our approach, according to the difficulty the player is facing, we propose that the agent choose actions with high or low expected return. For a given situation, if the game level is too hard, the agent does not choose the optimal action (the one with highest action value), but chooses progressively sub-optimal actions until its performance is as good as the player's. This entails choosing the second best action, the third one, and so on, until it reaches the player's level. Similarly, if the game level becomes too easy, it will choose actions whose values are higher, possibly until it reaches the optimal policy. Figure 1 illustrates a possible configuration for an agent acting at its second best performance level for a tabular representation of the learned knowledge. However, this sub-optimal challenge-sensitive strategy can also be applied to any other implementation of the action-value function. It is also possible to mix the challenge-sensitive action selection with softmax rules [16], in which each action is associated to a probability of being choose based on its value difference for the desired action-value level.

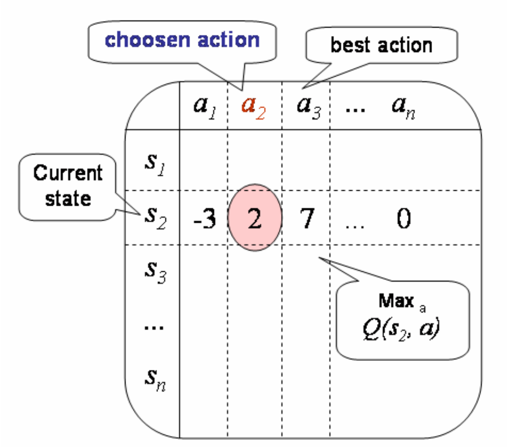

Figure 1: An agent acting at the second best performance level.

Our approach uses the order relation naturally defined in a given state by the action-value function, which is automatically built during the learning process. As these values estimate the individual quality of each possible action, it is possible to have a strong and fast control on the agent behavior and consequently on the game level.

A possible source of instability in this action selection mechanism is that the values of the action-value function express the expected reward obtained by the current and all subsequent actions, discounted by the parameter γ. So, if the agent gets delayed reward, it would need to persist in the chosen policy in order to get the desired return. It means that we need to define a cycle to determine when the agent will analyze, and possibly change, its performance level. In fact, this cycle should be the same as the one used by the challenge function to measure the difficulty the player is facing.

In this challenge-sensitive action selection mechanism, the agent periodically evaluates if it is at the same level of the player, through the challenge function, and according to this result, maintains or changes its performance level. The agent does not change its level until the next cycle. This evaluation cycle is strongly tied to the particular environment, in which the agent acts, depending, in particular, on the delay of the rewards. If the cycle is too short, the agent can exhibit a random behavior; if the cycle is too long, it will not match the player evolution (or regression) fast enough.

It is important to notice that this technique changes only the action selection procedure, while the agent keeps learning during the entire game. It is also worthwhile to stress that this action selection mechanism coupled with a successful off-line learning phase (during the bootstrap phase), can allow the agent to be fast enough to play at the user level at the beginning of the game, no matter how experienced he or she is.

5. Case study

5.1. Game description

As a case study, the approach was applied to Knock'Em [2], a real-time fighting game where two players face each other inside a bullring. The main objective of the game is to beat the opponent. A fight ends when the life points of one player (initially, 100 points) reach zero, or after 1min30secs of fighting, whatever comes first. The winner is the fighter which has the highest remaining life at the end. The environment is a bidimensional arena in which horizontal moves are free and vertical moves are possible through jumps. The possible attack actions are to punch (strong or fast), to kick (strong or fast), and to launch fireballs. Punches and kicks can also be performed in the air, during a jump. The defensive actions are blocking or crouching. While crouching, it is also possible for a fighter to punch and kick the opponent. If the fighter has enough spiritual energy (called mana), fireballs can be launched. Mana is reduced after a fireball launch and continuously refilled during time at a fixed rate.

5.2. Learning to fight

The fighters' artificial intelligence is implemented as a reinforcement learning task. As such, it is necessary to code the agents' perceptions, possible actions and reward signal. We used a straightforward tabular implementation of Q-learning, since, as we will show, although simple, this approach provided very good results. The state representation (agent perceptions) is represented by the following tuple:

$$S = (S_{agent}, S_{opponent}, D, M_{agent}, M_{opponent}, F)$$

S$_{\mathbf{agent}}$ stands for the agent state (stopped, jumping, or crouching). **S**$_{\mathbf{opponent}}$ stands for opponent state (stopped, jumping, crouching, attacking, jump attacking, crouch attacking, and blocking). **D** represents opponent distance (near, medium distance and far away). **M** stands for agent or opponent mana (sufficient or insufficient to launch one fireball). Finally, **F** stands for enemies' fireballs configuration (near, medium distance, far away, and no existence).

The agent's actions are the ones possible to all fighters: punching and kicking (strong or fast), coming close, running away, jumping, jumping to approximate, jumping to escape, launching fireball, blocking, crouching and standing still.

The reinforcement signal is based on the difference of life caused by the action (life taken out from opponent minus life lost by the agent). As a result, the agent reward is always in the range [-100, 100]. Negative rewards mean bad performance, because the agent lost more life than was taken from the opponent, while positive rewards are the desired agent's learning objective.

The RL agent's initial intelligence is built through offline learning against other agents. We used state-machine, random and another RL agent (self-learning) as trainers. The 3 agents trained with different opponents fought then against each other in a series of 30 fights. The resulting mean of life differences after each fight showed that the agents trained against the random and the learning agents obtained the best performances. Since the difference between these two latter agents is not significant, in the next section experiments, the agent that learned against a random agent is considered as the starting point for the online RL agent.

In all RL agents, the learning rate was fixed at 0.50 and the reward discount rate at 0.90, both for offline and online learning. This high learning rate is used because, as the opponent can be a human player with dynamic behavior, the agent needs to quickly improve its policy against him or her.

5.3. Fighting at the user's level

The action selection mechanism proposed was implemented and evaluated in Knock'em. The challenge function used is based on the life difference during the fights. In order to stimulate the player, the function is designed so that the agent tries to play better than him or her. Therefore, we empirically stated the following heuristic challenge function: when the agent's life is smaller than the player's life, the game level is easy; when their life difference is smaller than 10% of the total life (100), the game level is medium; otherwise, it is difficult.

The evaluation cycle used is 100 game cycles (or game loops). This value was empirically set to be long enough so that the agent can get sufficient data about the opponent before evaluating him or her, and short enough so that the agent quickly adapt to the player's behavior.

$$f = \begin{cases} easy,\ if \quad L(agent) < L(player) \\ medium,\ if \quad L(agent) - L(player) < 10 \\ difficult,\ otherwise \end{cases}$$

The implementation of the proposed action selection mechanism is summarized as follows. The agent starts the game acting at its medium performance level. As in Knock'em there are thirteen possible actions and so thirteen possible levels in our adaptive agent, it starts at the sixth level. During the game, the agent chooses the action which is the 6th highest value at the action-value function. After the evaluation cycle (100 game cycles), it evaluates the player through the previous challenge function. If the level is easy, the agent regresses to the 7th level; if it is difficult, it progresses to the 5th level; otherwise, it remains on the 6th level. As there are approximately 10 evaluations through a single fight, the agent can advance to the best performance level or regress to the worst one in just the first fight.

5.4. Experimental results

Since it is too expensive and complex to run experiments with human players, we decided to initially validate our adaptation approach comparing the performance of three distinct agents: a traditional state-machine (script-based agent), a traditional reinforcement learning (playing as well as possible), and the adaptive agent (implementing the proposed approach). Both reinforcement learning based agents (the traditional and the adaptive) were previously trained against a random agent. The evaluation scenario consists of a series of fights against different opponents, simulating the diversity of human players strategies: a state-machine (SM, static behavior), a random (RD, unforeseeable behavior) and a trained traditional RL agent (TRL, intelligent and with learning skills).

Each agent being evaluated plays 30 fights against each opponent. The performance measure is based on the final life difference in each fight. Positive values represent victories of the evaluated agent, whereas negative ones represent defeats. Figure 2 shows the state-machine (SM) agent performance against each of the other 3 agents. The positive values of the white line show that the agent can beat a random opponent, but the negative points show that sometimes the former loses against the latter. The black line shows that two state-machine fighters have a similar performance while fighting against each other. The negative gray points show that the traditional RL agent always beats the state-machine.

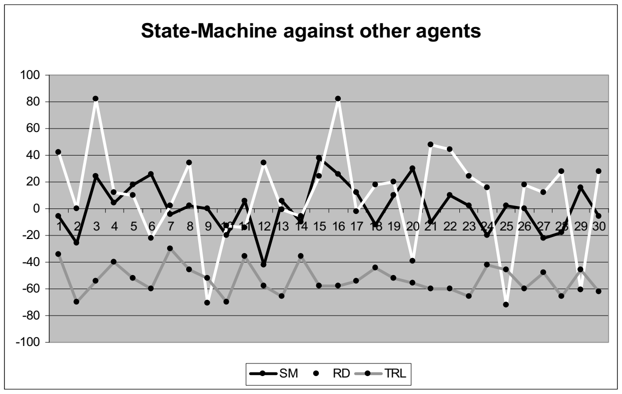

Figure 2: State-machine agent performance (SM = State Machine, RD = Random, TRL = traditional no adapting RL).

In each evaluation, testing hypotheses (p-value significance tests) are provided to check whether the mean of the differences of life in each set is different of zero, at a significance level of 5%. If the mean is significantly different from zero, then one of the agents is better than the other; otherwise, the agents must be playing at the same level. Table 1 summarizes these data and shows that the SM agent plays at the same level of other SM and a RD agent, but is significantly worse than a TRL agent.

Table 1: State-machine performance analysis

	Mean	Std. deviation	p-value	Difference is significant?
SM	1,20	18,32	0,72	No
RD	9,23	37,13	0,18	No
TRL	-52,73	10,92	0,00	Yes

Figure 3 and Table 2 show that the traditional RL agent beats quite easily the state-machine and the random players. As in the previous case (state-machine agent analysis) the performance is equivalent when two identical agents are fighting.

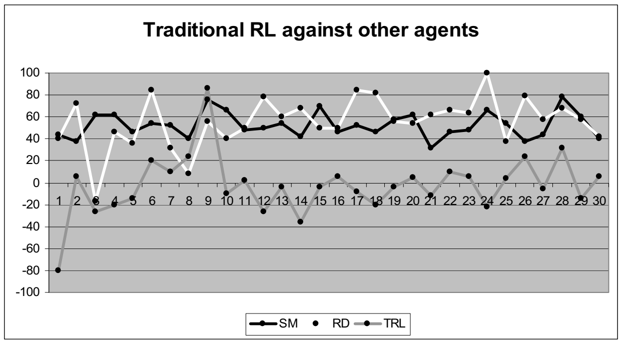

Figure 3: Traditional RL agent performance (SM = State Machine, RD = Random, TRL = traditional no adapting RL).

Table 2: Traditional RL performance analysis

	Mean	Std. deviation	p-value	Difference is significant?
SM	52,47	11,54	0,00	Yes
RD	55,47	23,35	0,00	Yes
TRL	-2,17	27,12	0,66	No

Figure 4 illustrates the adaptive RL agent performance. Although this agent has the same capabilities as traditional RL, because their learning algorithms and their initial knowledge are the same, the adaptive mechanism forces the agent to act at the same level as the opponent. The average performance shows that most of the fights end with a small difference of life, meaning that both fighters had similar performance. Table 3 confirms these results showing that the adaptive agent is equal to the three agents.

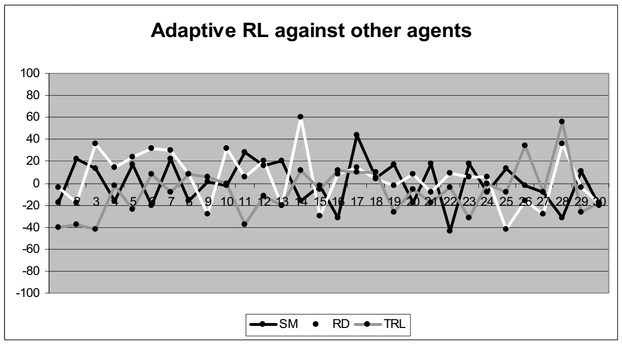

Figure 4: Adaptive RL agent performance (SM = State Machine, RD = Random, TRL = traditional no adapting RL).

Table 3: Adaptive RL performance analysis

	Mean	Std. deviation	p-value	Difference is significant?
SM	0,37	20,67	0,92	No
RD	4,47	23,47	0,30	No
TRL	-7,33	21,98	0,07	No

The adaptive RL agent (ARL) obtains similar performance against different opponents because it can interleave easy and hard actions, balancing the game level. Figure 5 shows a histogram with the agent actions frequency. The thirteen x-values correspond to all actions the agent can perform. The leftmost action is to the one with the highest value at the action-value function, while the rightmost is the one with lowest value. This way, the leftmost is the action which the agent believes to be the strongest one in each state and the rightmost is the action it believes to be the lighter one. The high frequency of powerful actions against the TRL agent shows that the ARL agent needed to play at an advanced performance level. The frequency of actions against the SM and RD agents shows that the ARL played in an easier level, around the 9th and 11th one.

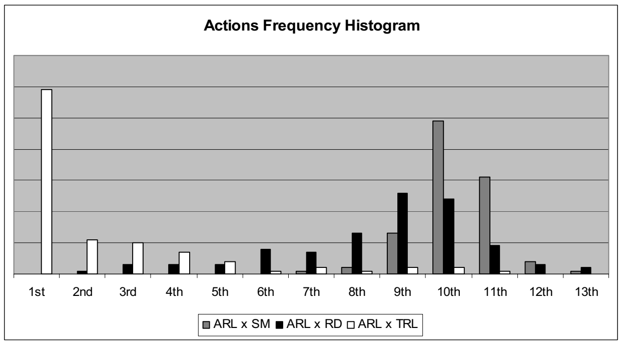

Figure 5: Histogram for the adaptive agent

6. Conclusions

The RL community has studied in detail the exploration versus exploitation dilemma [7]. An agent using a model-free algorithm like Q-Learning on a dynamic environment, should be constantly deciding whether to exploit the already learned knowledge, by choosing the actions with the highest estimates of expected return, or to explore new actions (possibly randomly chosen), to update these estimates (to learn more).

In this work, we introduce another dimension (challenge-sensitive) in RL action selection for scenarios involving a human user and an agent (or possibly various agents). According to this dimension, instead of choosing the best action (exploitation) or choosing a random action (exploration), which might not be believable, the agent analyses its own learned knowledge in order to choose actions which are just good enough to be challenging for the human opponent, whatever his or her level. This work presents thus an original approach to construct agents that dynamically adapt their behavior while learning in order to keep the game level adapted to the current user skills, a key problem in computer games and other applications requiring adaptive human-computer interactions, such as computer aided instruction applications.

The results obtained in a real-time fighting game indicate the effectiveness of our approach. In fact, although the adaptive agent could easily beat their opponents, the performance level is adapted so it plays close to the level of its opponent, interleaving wins and defeats. These results show that this innovative use of RL can provide a strong and fast control on the agent behavior and consequently on the game level.

We are now running systematic experiments with around 100 human players, who are playing against different types of agents, including our adaptive one, to check which one is really entertaining. The first results are encouraging, but the study is ongoing.

7. References

[1] Abramson, M., and Wechsler, H., "Competitive Reinforcement Learning for Combinatorial Problems", In *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, 2001, pp. 2333-2338.
[2] Andrade, G., Santana, H., Furtado, A., Leitão, A., and Ramalho, G., "Online Adaptation of Computer Games Agents: A Reinforcement Learning Approach". In *Proceedings of the 3rd Brazilian Workshop on Computer Games and Digital Entertainment*, Curitiba, 2004.
[3] Beck, J., Stern M., and Woolf B., "Using the Student Model to Control Problem Difficulty", In *Proceedings of Sixth International Conference on User Modeling*, Vienna, 1997, pp. 277-288.
[4] Demasi, P., and Cruz, A., "Online Coevolution for Action Games". In *Proceedings of The 3rd International Conference on Intelligent Games And Simulation*, London, 2002, pp. 113-120.
[5] Falstein, N., "The Flow Channel". *Game Developer Magazine*, May Issue, 2004.
[6] Hunicke, R., and Chapman, V., "AI for Dynamic Difficulty Adjustment in Games". *Challenges in Game Artificial Intelligence AAAI Workshop,* San Jose, 2004, pp .91-96.
[7] Kaelbling, L., Littman, M., and Moore, A., "Reinforcement Learning: A Survey", *Journal of Artificial Intelligence Research*, AAAI Press, 1996, pp. 4:237-285.
[8] Koster, R., *Theory of Fun for Game Design*, Paraglyph Press, Phoenix, 2004.
[9] Langley, P., "Machine Learning for Adaptive User Interfaces", *Kunstiche Intellugenz*, 1997, pp. 53-62
[10] Madeira, C., Corruble, V., Ramalho, G., and Ratitch, B., "Bootstrapping the Learning Process for the Semi-automated Design of a Challenging Game AI". *Challenges in Game Artificial Intelligence AAAI Workshop,* San Jose, 2004, pp. 72-76.
[11] Manslow, J., "Using Reinforcement Learning to Solve AI Control Problems", In *Steve Rabin, editor, AI Game Programming Wisdom 2,* Charles River Media, Hingham, MA, 2003.
[12] Nielsen, J., *Usability Engineerin,* Morgan Kaufmann, San Francisco, 1993.
[13] Merke A., and Riedmiller, M. "Karlsruhe Brainstormers - A Reinforcement Learning Approach to Robotic Soccer". *RoboCup 2001. RoboCup-2000: Robot Soccer World Cup IV*, London, 2001, pp. 435-440.
[14] Samuel, A., "Some studies in machine learning using the game of checkers II-Recent progress", *IBM Journal on Research and Development*, 1967, 11:601-617.
[15] Spronck, P., Sprinkhuizen-Kuyper, I., and Postma, E., "Difficulty Scaling of Game AI". In *Proceedings of the 5th International Conference on Intelligent Games and Simulation*, Belgium, 2004, pp. 33-37.
[16] Sutton, R., and Barto, A., *Reinforcement Learning: An Introduction*, The MIT Press, Massachusetts, 1998.
[17] Tesauro,. G., "TD-Gammon, a self-teaching backgammon program, achieves master-level play", *Neural Computation*, The MIT Press, Massachusetts, 1994, 6(2): 215-219.
[18] Watkins, C., and Dayan, P., "Q-learning", Machine Learning, 1992, 8(3):279–292.
[19] Wiegand, R., Liles, W., and Jong, G., "Analyzing Cooperative Coevolution with Evolutionary Game Theory", In *Proceedings of the 2002 Congress on Evolutionary Computation*, IEEE Press, Honolulu, 2002, pp. 1600-1605.