2011 Conference on Technologies and Applications of Artificial Intelligence

Evolving Intelligent Mario Controller by Reinforcement Learning

Jyh-Jong Tsay
Department of Computer Science and Information Engineering,
National Chung Cheng University,
Chiayi, Taiwan, Republic of China
E-mail: tsay@cs.ccu.edu.tw

Chao-Cheng Chen
Department of Computer Science and Information Engineering,
National Chung Cheng University,
Chiayi, Taiwan, Republic of China
E-mail: ccch96m@cs.ccu.edu.tw

Jyh-Jung Hsu
Department of Computer Science and Information Engineering,
National Chung Cheng University,
Chiayi, Taiwan, Republic of China
E-mail: dcjhsu@gmail.com

Abstract—**Artificial Intelligence for computer games is an interesting topic which attracts intensive attention recently. In this context, Mario AI Competition modifies a Super Mario Bros game to be a benchmark software for people who program AI controller to direct Mario and make him overcome the different levels. This competition was handled in the IEEE Games Innovation Conference and the IEEE Symposium on Computational Intelligence and Games since 2009. In this paper, we study the application of Reinforcement Learning to construct a Mario AI controller that learns from the complex game environment. We train the controller to grow stronger for dealing with several difficulties and types of levels. In controller developing phase, we design the states and actions cautiously to reduce the search space, and make Reinforcement Learning suitable for the requirement of online learning.**

Keywords-games; Super Mario Bros; Game AI; Reinforcement Learning

I. INTRODUCTION

Computer games play a more and more important role in our lives. In this case, more and more researches and conferences concentrate on this issue. One of the most famous digital games is Super Mario Bros [1]. Super Mario Bros is a kind of platform games [2] that requires a player to control the game character to move or jump from a platform to another platform, and sometimes to cross over obstacles or trap until reaching the goal or satisfying some requests.

Computer games often have large or sometimes continuous state and action spaces. These make digital games become good surroundings for investigation in Artificial Intelligence (AI) and Machine Learning (ML). The ways people used to play games are usually interesting issues for AI and ML to explore. For this reason, conferences have been held to run competitions to evolve AI robots for game playing in recent years. One of these competitions is Mario AI Competition [3] which was handled in the IEEE Games Innovation Conference (ICE-GIC) and the IEEE Symposium on Computational Intelligence and Games (CIG) since 2009. The Mario AI Competition provides benchmark software every year for competitors who join the competition. They build their own Mario AI agent on the benchmark and estimate their agents' performance by using the evaluation system in the benchmark.

Reinforcement Learning (RL) [4] is one of the widely studied methods in AI and ML, which learns from interaction with the environment. RL is one of the promising approaches to implement intelligent game agents that can adapt to the behavior of a player, or to dynamic changes in the game world. In this paper, we study the application of RL to construction of Mario AI Controllers which have the capability of learning from complex game environments. The major issue in this study is how to reduce the huge number of state and action pairs to a practical level to make RL feasible. We achieve this by cautiously simplifying game environment states, and grouping Mario's actions. We use Q-Learning to online evolve a state-to-action decision strategy which aims to maximize the reward, and learns even after the game starts. Experiment over the evaluation used in 2009 Mario AI Competition shows that our RL agent achieves very good performance which is better than any learning-based approaches submitted to 2009 Competition.

The rest of this paper is organized as follows. Section 2 and 3 review some related work and the benchmark. Section 4 gives basics of Q-Learning. Section 5 gives our design of Mario Controller. Section 6 gives experimental result. Section 7 gives conclusion and future remarks.

II. RELATED WORKS

In this section, we review some related work. The 2009 Mario AI Competition [3] introduced the benchmark [5] used in the competition, and the rules for competitors to design agents. The most surprising result of the competition is that top three agents of the competition are implemented by A* search algorithm [6], outperforming agents by rule-based, genetic programming or neural network algorithms. However, learning-based approaches have the advantages of evolving and adapting the controllers as the environment gets more and more complicated.

REALM (Rule-Based Evolutionary Computation Agent that Learns to Play Mario) by Bojarski and Congdon [8] is a rule-based system evolved by genetic algorithms. REALM has both hard-coded and learned version of rule-based agents to control Mario. In [13], Mohan and Laird present a controller based on neural networks evolved by generic programming.

Reinforcement Learning [4] motivated from the observation of human and animal behaviors that learn from interaction with the environment has been widely used in AI. SARSA (State-Action-Reward-State-Action) [9] is an online variation of RL in which the reward is given right after the action is chosen. One of the main issues in RL for games is the high dimensionality of state and action spaces.

Modular Reinforcement Learning [10] is one of the approaches to solve the massive state-action pairs' problem in RL. It reduces the heavy search space by splitting the states into smaller tasks which have its own reinforcement learner. Soar-RL[13] implements a hierarchy of actions to reduce action space, and converts continuous distance between Mario and other entities to attributes like *isthreat* and *isreachable* through elaboration rules. In this paper, we reduce the state and action spaces by cautiously simplifying game environment states, and grouping Mario's actions.

III. Benchmark

The benchmark is a modified version of Infinite Mario Bros [11] which is a modified version of Super Mario Bros. Infinite Mario Bros is a free open source based on Java, it was programmed by Markus Persson. Mario AI Competition improves the Infinite Mario Bros' game engine and the default level generator to make the game more suitable for competing and machine learning.

We will explain the benchmark by separating it into the following five components: *A. Control, B. Mario, C. Enemies, D. Environment, E. Available APIs and Important Constraints.*

A. Control

Mario is controlled by six buttons in the benchmark, and has the same type of control mode with Super Mario Bros. These six buttons are UP, DOWN, LEFT, RIGHT, JUMP and SPEED. Although the UP direction is unused in the benchmark, there are still five buttons could be composed. In other words, we have $2^5 - 8 = 24$ possible actions if we eliminate the conflicting button that may be pressed (e.g. press the left button and right button in the same time).

B. Mario

Mario is the main character in the game competition. The main target is controlling Mario to reach each goal of each level which is located at the most right side of levels. Mario will lose a life when he falls into a gap, level's time is up or gets hurt when he is in small size. If Mario loses a life, the game is over.

Mario has three states: small, big and fire. In fire mode, Mario gets a skill that shoots fireball to beat enemies. In big mode and small mode, Big Mario is larger than Small Mario, but larger size makes Big Mario has more chance to get hurt.

All Mario's modes can beat the enemies by stomping on their head or using the shell to collide them. Mario produces a shell by stomping on the turtle-like enemy. But if Mario touches enemies instead of stomping on their head, Mario will get hurt. When Fire Mario gets hurt, he degrades to Big Mario. When Big Mario gets hurt, he degrades to Small Mario. If Small Mario gets hurt, he will lose a life. Fortunately, Mario could restore his health point by reaching some items. We will describe the items in Environment phase.

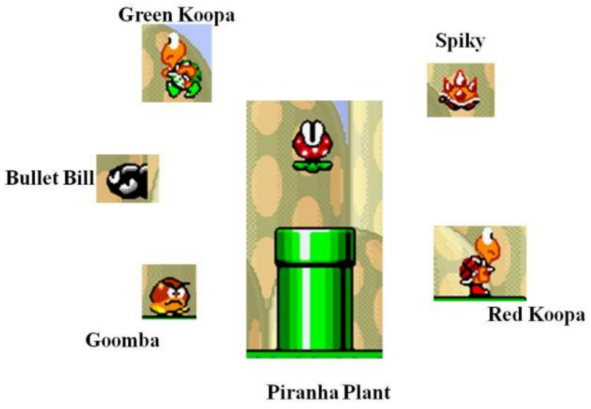


Figure 1. Mario's Enemies

C. Enemies

There are many kinds of enemies in the Mario world. Each enemy has dissimilar dangers to Mario. Fig. 1 shows the different types of enemies. The features of enemies are expressed as following:

- **Goomba:** Mario can beat Goomba by stomping on them, shooting a fireball or dropping a shell to collide him.
- **Green & Red Koopa:** If Green Koopa walks from a higher platform to a lower platform, he would cross the cliff edge and arrive the lower ground. Red Koopa is almost the same like Green Koopa but the Red Koopa doesn't cross the cliff edge. Mario can beat Koopa like the ways he does to Goomba. If Mario stomps on Koopa, it becomes a shell on the ground. Shell is a strong weapon that hurts all of the enemies but it also hurts Mario.
- **Spiky:** Spiky has a hard and spiked carapace. He cannot be beat by stomping and fireball The only weapon that defeats Spiky is the Koopa Shell.

The above enemies each have a winged version. Winged enemies' weaknesses are the same as the overground version. If Mario stomps on them expect the winged Spiky, they just transform to the overground version and Mario has to beat it again. There are more enemies without the winged version which are listed below.

- **Bullet Bill** which is shot from the cannon. Bullet Bill flies in the air, and is not affected by the gravity. He can be stomped and collided by shell but immune to fireballs.
- **Piranha Plant:** Piranha Plant jumps from a green water pipe and goes back to pipe after a while. Mario beats him by shooting fireballs and dropping shells. If Mario stomps on him, he would be bitten by the plant. Mario may jump on the pipe when

Piranha Plant goes back, this causes that Piranha Plant hides in the pipe and not attacks Mario.

We know that there are 4*2 + 2 = 10 kinds of enemies will attack Mario now. If we add the shell into them, eleven dangerous creatures appear in Mario world.

D. Environment

Objects that Mario sees in surroundings are also important elements in our Mario AI agent design. Four types of bricks occur in the benchmark. Mario can stand on the bricks if they exist.

If Mario is not in fire mode, we could try to produce some power-up items by breaking bricks. These items are very important if Mario is in Small state or in Sig state. Power-up items can heal Mario and give some ability to Mario. Here are the power-up items:

- **Mushroom:** Upgrade Mario from small mode to big mode and big Mario could bump the bricks. If big Mario and fire Mario reach it, nothing happens.
- **Fire Flower:** Upgrade Mario from big mode to fire mode. In fire mode, Mario can bump the bricks, and shoot fireballs. If fire Mario reaches it, nothing happens.

Fig. 2 shows the relativity between Mario and power-up items. An additional item shows in following:

- **Coin:** Adds bonus to Mario's score.
- The last main points in this phase are landforms and obstacles. Bad landforms and obstacles make Mario get hurt easily. The worst situation is that Mario dies immediately and loses the game. The following is the main landforms and obstacles that have to pay attention.
- **Hill:** In this landform, Mario passes through the field at lower part and doesn't get stuck by it. Mario can jumps on it like standing on the ground.
- **Gap:** Gap is the most dangerous landform to Mario. We have be caution to cross gaps. If Mario falls into it, he loses a life immediately!
- **Cannon:** Cannon is a black tall column on the ground generally and it never moves. This obstacle shoots limitless Bullet Bills after a period of time and it is usually very tall to block Mario's path.
- **Pipe:** Its appearance is a Is two bricks width tube and usually not too tall. Some of them hide Piranha Plant and some are not. We can observe the pipe in a small time to check there is Piranha Plant hiding in it or not.

We can see that the objects in the environment may help Mario or damage Mario. How to use these items to support Mario reaching the goal is also the key of our research.

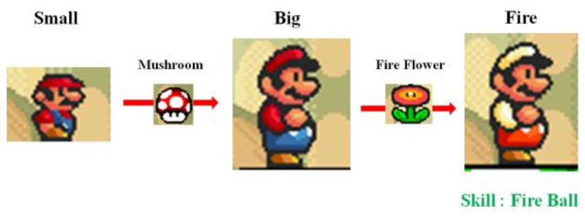


Figure 2. The process of Mario's power-up

E. Available APIs and Important Constraints

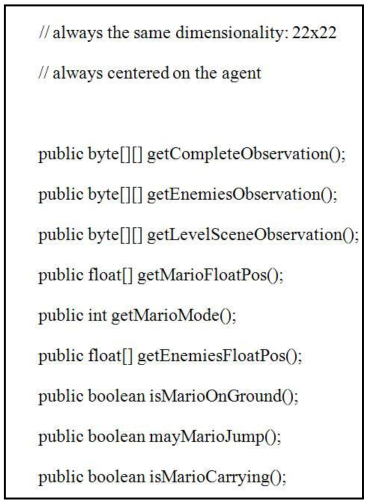
This benchmark provides some available APIs in the Environment Java interface to receive information from the Mario world for controllers in the Mario AI Competition. These APIs are shown in Fig. 3.

We get the information from APIs in Environment interface. The getCompleteObservation() method returns enemies' part, landforms' part and obstacles' part in the game environment or we can choose a simple version like getEnemiesObservation() or getLevelScenebservation() to develop our controller. Zero presents a passable block and the other number has its own meaning in these methods.

There are important and critical constraints we must understand and obey them.

First, we have to implement our controller by using the form of Agent interface in the benchmark. The competition receives this agent and compares to the other agents. We cannot change anything in the other code field of the benchmark like modifying the game engine to earn more score. Otherwise, we will be disqualified.

Second, the frequency of the benchmark updating game environment is 24 fps. This means that the benchmark takes an action from the controller every about 42 milliseconds (the time consumes to update one frame) by force. If we overtake this time in a level, we will be disqualified.



```
// always the same dimensionality: 22x22

// always centered on the agent


public byte[][] getCompleteObservation();

public byte[][] getEnemiesObservation();

public byte[][] getLevelSceneObservation();

public float[] getMarioFloatPos();

public int getMarioMode();

public float[] getEnemiesFloatPos();

public boolean isMarioOnGround();

public boolean mayMarioJump();

public boolean isMarioCarrying();
```

Figure 3. Some APIs in Environment Java interface of the benchmark which include Mario's observation of objects and

enemies in Mario world and these APIs are the main tools for controller design.

IV. Q-LEARNING

We use the Q-Learning [12] algorithm in our study. Q-Learning is a family temporal difference algorithm of RL. Temporal difference algorithm updates its value according to previous experience and reinforcement signal received from the environment in the same state with current state.

A. Q-Learning

Q-Learning considers the learning environment to a state machine. The learning process always stays in some states at any time in Q-Learning algorithm. The environment's details in state will affect the performance of learning directly and Q-Learning has to give each of the different states a unique state number for setting values.

The states and actions are combined to state-action pairs in Q-Learning. It saves every state-action pair's Q-value into the Q-table. Q-value is expressed as $Q(s, a)$, s stands for state and a stands for actions. Q-table contains the Q-value which is updated by receiving reward that was given after action is chosen in the particular state over time with rule (1). There are four main factors of Q-Learning algorithm represented by following: current state, chosen action, reward and future state. When the agent receives a reward, the algorithm updates the Q-value by following rule:

$$Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha(r + \gamma\max(Q(s_{t+1}, a_{t+1}))) \qquad (1)$$

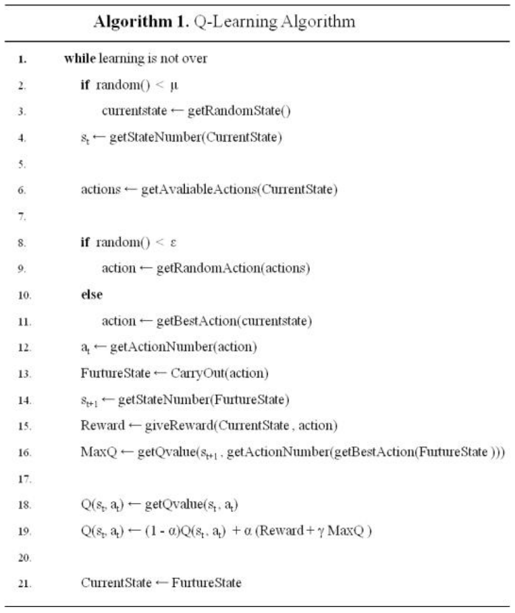
After an action is taken, the Q-Learning algorithm thinks that the environment changes from current state to the future state. $Q(s_t, a_t)$ stands for the current state's Q-value and $Q(s_{t+1}, a_{t+1})$ stands for the future state's Q-value. The $\max(Q(s_{t+1}, a_{t+1}))$ stands for the highest Q-value in future state. $Q(s_t, a_t)$ saves back into Q-table and overwrite it after mixing with reward r and the highest Q-value of the future state.

Q-Learning doesn't care about the future state's Q-value and just chooses the maximum to add into the old state's Q-value. The following are parameters in Q-Learning.

- **α:** the learning rate in the range [0,1]. This value controls the percentage of the feedback and the old state's Q-value that mix to each other.
- **γ:** the discount rate in the range [0,1]. This value controls how much importance of future Q-value we treat.
- **r:** the reward given after action was taken. This parameter is very important to account into algorithm. Low quality of reward's setting would direct Q-Learning algorithm losing its learning ability!

Fig. 4 shows the complete Q-Learning algorithm. In the algorithm, μ and ε are the exploration rates explained below.

- **μ:** This rate controls the probability that the agent starts random exploation by taking a random state. It is in the range [0,1].
- **ε :** This rate controls the probability that the agent take a random action in current state. It is in the range [0,1].



Algorithm 1. Q-Learning Algorithm

```
while learning is not over
    if random() < μ
        currentstate ← getRandomState()
    $s_t$ ← getStateNumber(CurrentState)

    actions ← getAvaliableActions(CurrentState)

    if random() < ε
        action ← getRandomAction(actions)
    else
        action ← getBestAction(currentstate)
    $a_t$ ← getActionNumber(action)
    FurtureState ← CarryOut(action)
    $s_{t+1}$ ← getStateNumber(FurtureState)
    Reward ← giveReward(CurrentState , action)
    MaxQ ← getQvalue($s_{t+1}$ , getActionNumber(getBestAction(FurtureState )))

    $Q(s_t, a_t)$ ← getQvalue($s_t$ , $a_t$)
    $Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha$ (Reward + $\gamma$ MaxQ )

    CurrentState ← FurtureState
```

Figure 4. Q-Learning Algorithm

In the algorithm, agent begins from a start state or begins form the random state, then he chooses the action with the highest Q-value or selects the random action. After performing the action, the agent reaches new state and gets the rewards. He uses the highest Q-value in new state and rewards to overwrite the old Q-value in previous state according to learning rate and discount rate. Finally, he starts from the new state and does the above things again until learning is over.

B. Q-Learning's Disadvantage

Q-Learning algorithm is simple to implement but it has a critical weakness. Q-table is a main component in the Q-Learning for storing experiences in learning process. If we use Q-Learning to learn from a complex environment, it may have a very large number of state-action pairs. This makes the Q-table consume a lot of memory space in the operate system or even the operate system doesn't has enough memory space to store the Q-table. It also has to spend much time for saving or loading the Q-values! This problem will very seriously affect the performance and quality of learning.

V. MARIO CONTROLLER DESIGN USING Q-LEARNING

After above discussion, we know the problem in Q-Learning that it is not good when environment is complex because the state-action pairs would be very large, and decrease the performance of learning. Unfortunately, this problem occurs in this study.

We know there eleven creatures, three items, two bricks, landforms and obstacles would be presented to the game environment. Even more, we have a visual field of 22*22 sensor grids and each grid Mario has 24 possible actions to perform. This would cause a massive search spaces that be produced in Mario world and have difficulty for Q-learning. The possible state-action pairs show blow:

$$(11+3+2+3)^{(22*22-1)} * 24$$

It is an incredible number. The feasible way is to use lower sensors grid for reducing state-action pairs. But if we use lower sensors in environment this would also decrease the quality of learning.

For this problem, we refine the elements in states and combine the actions to the task of action sets for reducing search space. We construct two versions of Q-Learning controllers. They are called Q-LearnerV1 and Q-LearnerV2.

A. State and Action Spaces

In Q-LearnerV1, we choose 10 attributes to compose the states in environment. This following is the list of elements:

- **Mario Mode:** Display Mario's size in the state, 0 represents small, 1 represents big and 2 represents fire.
- **Higher enemies near front side:** This attribute is true meaning that there is an enemy existing in the right side of Mario within the given range and his position is higher than Mario. False means no enemies within the given range.
- **Higher enemies near backside:** This attribute is true meaning that there is an enemy existing in the left side of Mario within the given range and his position is higher than Mario. False means no enemies within the given range.
- **Lower enemies near front side:** This attribute is true meaning that there is an enemy existing in the right side of Mario within the given range and his position is lower than Mario or equal to him. False means no enemies within the given range.
- **Lower enemies near backside:** This attribute is true meaning that there is an enemy existing in the left side of Mario within the given range and his position is lower than Mario or equal to him. False means no enemies within the given range.
- **Brick Exist:** True if there exists bricks within the given range of Mario. If no bricks, output false.
- **May Upgrade:** If there exists a mushroom or a fire flower, this attribute is true. Otherwise, it is false.
- **Near Gap:** If there is a gap within the given range of Mario, this attribute is true. Otherwise, it is false.
- **On Gap:** True if Mario is jumping across a gap of falling in the gap. False means no gaps under Mario.
- **Enemy Type:** This attribute is 4-ary-valued, 0 represents the nearest enemy doesn't fear stomp and fireball. 1 represents the nearest enemy could be stomped. 2 represents the nearest enemy could beat by fireball. 3 represents the nearest enemy could beat by stomping and fireball.

We also combine actions to action sets for constructing main action tasks. Each action task executes its own plan for Mario and outputs a sequence of actions. These action tasks support Mario the reach the goal.

- **Stomp Attack Subtask:** Take charge of attacking enemies by stomping.
- **Fireball Attack Subtask:** Take charge of attacking enemies by fireball.
- **Shell Attack Subtask:** Take charge of attacking enemies by shell.
- **Upgrade Task:** This task controls Mario to reach power-up item or bump the bricks for trying producing power-up items.
- **Pass Through Task:** This task controls Mario to cross the landforms and obstacles to get right side in the scene and try not falling into gaps.
- **Avoidance Task:** Take charge of avoiding enemies and not getting hurt. Mario may wait or find a safe way to escape enemies.

The first three attack subtasks in above we write them into a big Attack Task. In our Q-Learning, each state has all of the tasks and each task has its own Q-value. In Q-LearnerV1, the state-action pairs have reduced to the following state-action task pairs:

$$3 * 2^8 * 4 * 6 = 18432 \text{ possible state-action task pairs}$$

In Q-LearnerV2, we use 9 attributes to compose the states in environment. The first 9 attributes are the same with Q-LearnerV1 and we eliminate the Enemy Type attribute. We don't split the Attack Task into subtasks. All stomping, shooting fireball and dropping shells strategies are in one Attack Task now. The other tasks are the same with Q-LearnerV1. Q-LearnerV2 doesn't learn the attack strategies, and all attack plans are provided by the Attack Task. All plans that deal with enemies are written in the task.

The number of state-action pairs in Q-LearnerV2 is reduced to 3072 as calculated below.

$$3 * 2^8 * 4 = 3072 \text{ possible state-action task pairs}$$

One of the critical constraints in section III is that controllers have to produce an action before the time limit is arrived in about 42 milliseconds. The main reason why we use Q-Learning is that it is simple and fast. It doesn't consume much time for computing the Q-value. As a result, our controller could have more time to evaluate the action produced from the action task to Mario.

B. Rewards and Parameters

In general, if Mario gets hurt or dies when executes a particular task, the task is given a big penalty. In the opposite side, if Mario reaches a power-up item when Mario is not in fire state, the task receives a big reward. If Mario arrive the goal, the action task receives a massive reward.

In Q-LearnerV1, rewards are set up as follows.

- **Attack Subtasks:** Reward is given if the numbers of enemies beat by stomping, fireball and shell raise after this task is over.
- **Upgrade Task:** If Mario reaches power-up item, he get a big reward. If there is no power-up item exists, Mario try to bump the brick. Mario get a little reward by bump each brick. However, if Mario reaches any power-up item when he is in a fire state. We give a penalty to him because we consider this is a useless work. We also give a penalty to Mario if he uses this task when no brick and no power-up item exists
- **Pass Through Task:** Reward is given when Mario moves a fixed distance to the right side.
- **Avoidance Task:** Reward is given when Mario moves a fixed distance and stays in a safety place.

In Q-LearnerV2, we set up the rewards almost the same but combining the settings of every attack subtasks into a single reward setting.

The learning rate in this study is 0.3. The discount rate is 0.01 because we consider the relativity from current state to future state is very little. The random state exploration rate is zero because Mario could not start with a random state in the benchmark. We want Mario to try more actions in the learning process so we set random action exploration rate with a value of 0.5.

VI. Experiment

We train our Q-LearnerV1 and Q-LearnerV2 with random seeds in the level generator. The level difficulties we set are as follows. First 500 levels have difficulties from 1 to 10, and the remaining 4500 levels have difficulties from 1 to 100. If the difficulty exceeds 100, it restarts from 1.

After training in 5000 levels, we compare our controllers to the scores in Mario AI Competition @ CIG 2009 with the same environment parameters. The results are presented in Table 1.

In Table 1, our Reinforcement Learning controllers perform much better than all rule-based and learning-based algorithms, although they are still worse than A* search algorithms. Furthermore, Q-LearnerV2 performs better than Q-LearnerV1. But compared to A*, RL controllers can improve their behaviors and learn adaptive actions stage by stage as time goes by. In addition, we observe that Mario using RL controllers has more attack tendency to enemies and can heal himself by searching the power-up items.

Fig. 5 shows the learning curve of Q-LearnerV1 and Q-LearnerV2 and gives the levels that the controllers could overcome on the benchmark with the same environment parameters of Mario AI Competition @ CIG 2009 every 100 training times. Note that both of them improves their performance as training times increase. We also observe that, in most cases, Q-LearnerV2 passes more levels, and adapts to the game environment faster than Q-LearnerV1. Q-LearnerV2 is a better choice than Q-LearnerV1. But we also observe that the variations of Q-LearnerV1 and Q-LearnerV2's learning curve are very similar to each other if the learning times are big enough.

VII. Conclusions and Future Work

In this paper, we show how to develop Mario AI controllers using Reinforcement Learning. We show how to reduce state and action spaces to a practical level to make RL feasible. The kernel of our controllers is the definition of the attributes in states and the higher-level action tasks. Experiment shows RL Mario Controller performs very well. In the future, we will continue to improve the controller to learn from interaction with more complex environments such as those announced for competition in 2010 and 2011.

Competitor	approach	progress	levels	time left	kills	mode
Robin Baumgarten	A*	46564.8	40	4878	373	76
Peter Lawford	A*	46564.8	40	4841	421	69
Andy Sloane	A*	44735.5	38	4822	294	67
Q-LeanerV2	RL	34021.7	30	3591	279	58
Q-LeanerV1	RL	30840	28	3534	206	54
Trond Ellingsen	RB	20599.2	11	5510	201	22
Sergio Lopez	RB	18240.3	11	5119	83	17
Spencer Schumann	RB	17010.5	8	6493	99	24
Matthew Erickson	GP	12676.3	7	6017	80	32
Douglas Hawkins	GP	12407.0	8	6190	60	37
Sergey Polikarpov	NN	12203.3	3	6303	67	38
Mario Perez	SM, Lrs	12060.2	4	4497	170	23
Alexandru Paler	NN, A*	7358.9	3	4401	69	43
Michael Tulacek	SM	6571.8	3	5965	52	14
Rafael Oliveira	RB	6314.2	1	6692	36	9
Glenn Hartmann	RB	1060.0	0	1134	8	71
Erek Speed	GA	out of memory				

Table 1. Scores of the algorithms in CIG 2009 competition, Q-LearnerV1 and Q-LearnerV2. RL: Reinforcement Learning, RB: Rule-Based, GP: Genetic Programming, NN: Neural Networks.

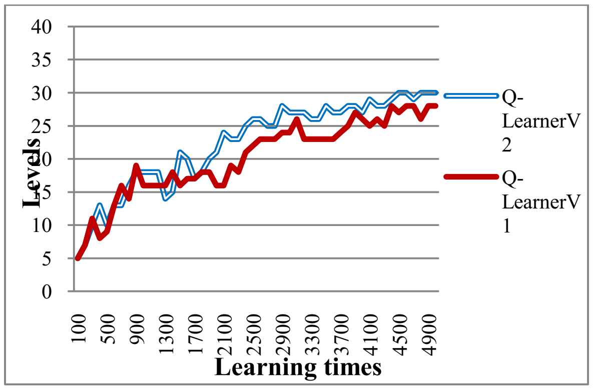


Figure 4. Learning curve of Q-LearnerV1 and Q-LearnerV2

References

[1] Super Mario Bros. [Online]. Available: http://en.wikipedia.org/wiki/Super_Mario_Bros.

[2] Platform game. [Online]. Available: http://en.wikipedia.org/wiki/Platform_game

[3] J. Togelius, S. Karakovskiy, and R. Baumgarten, *The 2009 Mario AI Competition*, in IEEE Congress on Evolutionary Computation, 2010.

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, 1998.

[5] J. Togelius, S. Karakovskiy, J. Koutn´ık, and J. Schmidhuber, *Super Mario Evolution*, in CIG'09: Proceedings of the 5th international conference on Computational Intelligence and Games. Piscataway, NJ, USA: IEEE Press, 2009, pp. 156–161.

[6] P. Hart, N. Nilsson, and B. Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100–107, 1968.

[7] I. Millington and J. Funge, *Artificial Intelligence for Games.* Morgan Kaufmann Pub, 2009.

[8] Bojarski and C. B. Congdon, *REALM: A Rule-Based Evolutionary Computation Agent that Learns to Play Mario*, Proceedings of the 6th international conference on Computational Intelligence and Games. Dublin, Ireland, IEEE Press, 2010, pp. 83–90

[9] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*, Engineering Department, Cambridge, University, 1994

[10] C.J. Hanna, R.J. Hickey, D.K. Charles and M.M. Black, *Modular Reinforcement Learning Architectures for Artificially Intelligent Agents in Complex Game Environments*, Proceedings of the 6th international conference on Computational Intelligence and Games. 2010, pp. 380–387

[11] Infinite Mario bros. [Online]. Available: http://www.mojang.com/notch/mario/index.html

[12] C. Watkins, *Learning from Delayed Rewards*, Cambridge University, England, 1989.

[13] S. Mohan, J. E. Laird, *Relational Reinforcement Learning in Infinite Mario*, Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10).

[14] A* search algorithm. [Online]. Available: http://en.wikipedia.org/wiki/ A*_search_algorithm