Training Pac-Man bots using Reinforcement Learning and Case-based Reasoning *

Fernando Domínguez-Estévez, Antonio A. Sánchez-Ruiz, and Pedro Pablo Gómez-Martín

Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid (Spain)
fedomi01@ucm.es, antsanch@ucm.es, pedrop@fdi.ucm.es

Abstract. Video games are an interesting field of study for many artificial intelligence researchers, since many different AI methods can be studied and tested with them, and later those investigations can be applied to many other situations. In this paper we use case based reasoning and reinforcement learning principles to train bots to play the Ms. Pac-Man vs. Ghosts game. In particular, we use the well-known Q-learning algorithm but replacing the Q-table with a case base. The use of cases allows us to deal with rich game state representation and inject domain knowledge in both the retrieval and the adaptation stages. Our initial experiments show that we can train bots either to reach high scores or to survive for a long time. However, the combination of both goals seems to be a more challenging problem.

Keywords: Reinforcement Learning, Case-based Reasoning, Pac-Man, video games

1 Introduction

Video games are an interesting field of study for many Artificial Intelligence (AI) researchers, since they provide complex but controlled environments in which many AI techniques can be studied and tested. Video games are also very interesting to compare different AI techniques and learn their strengths and weaknesses.

In this work we use one of the most popular video games of all time: Ms. Pac-Man vs. Ghosts [14]. Although Pac-Man may seem simple compared to some other kind of games, its special features make it a perfect target for AI researchers. Firstly, this game is simple enough to be quickly understood and it does not require a very powerful machine to be run. On the other hand, in spite of its simplicity, lots of different strategies can be applied while playing, some of them more intelligent than others, so it is the task of AI techniques to discover the best strategies. In fact, Ms. Pac-Man vs. Ghosts has been used in different

* Supported by Spanish Ministry of Economy, Industry and Competitiveness under grant TIN2014-55006-R

AI competitions during the last years [5] and it is going to be used again this year at this year at the CIG conference [13].

There are two main approaches to train automatic bots to play videogames without human intervention: Genetic Algorithms (GA) [6] and Reinforcement Learning (RL) [11]. Although both approaches learn how to play by playing thousand of games and trying different behaviors, they require different information from the game simulation and they explore the space of possible solutions using very different approaches. GA work with populations of solutions that are combined and mutated to produce new generations of better solutions. The probability of each individual to survive and be selected for the next generation is proportional to its fitness or how well it plays the game. RL algorithms, on the other hand, learn policies that associate "good" actions to execute in different game states based on rewards obtained during the game simulation.

In this paper we use case-based reasoning and reinforcement learning principles to train bots to play the Ms. Pac-Man vs. Ghosts game. In particular, we use the well-known Q-learning algorithm [12] but replacing the Q-table with a case base. The use of cases allows us to deal with rich game state representation and inject domain knowledge in both the retrieval and the adaptation stages.

The rest of the paper is organized as follows. Section 2 briefly describes the video game used in our experiments. Section 3 describes the foundations of the techniques we use for learning: Case-based Reasoning and Reinforcement Learning. Section 4 explains the decisions we made to implement our agent in the particular context of a Pac-Man game. Section 5 explains the experiments performed and the limitations that we found. Finally, the paper closes with related work, conclusions and future work.

2 Ms. Pac-Man vs. Ghosts AI

Ms. Pac-Man is an arcade game released in 1981 as an unauthorized copy of the original Pac-Man game. Such was its popularity, that Namco decided to make an official version of this game. It is very similar to the original game, since the goal of the game is still to survive in a maze, escaping from four ghosts while picking pills and fruits that give points to the player.

The version of the game that we use, Ms. Pac-Man vs. Ghosts AI[1] (Figure 1), is an implementation in Java designed to test different AI techniques. The behavior of both Pac-Man and the Ghosts can be specified implementing *controller* classes that make decisions on the next move (left, right, up or down) depending on the game state. This framework has been used in different Pac-Man AI competitions and provides some example bots implementing simple behaviors.

In our experiments we use the default controller for the ghosts that tries to reproduce the behavior of the ghosts in the original game. It is interesting to note that each one of the 4 ghosts uses a different heuristic to decide how to move.

[1] `https://github.com/kefik/MsPacMan-vs-Ghosts-AI`

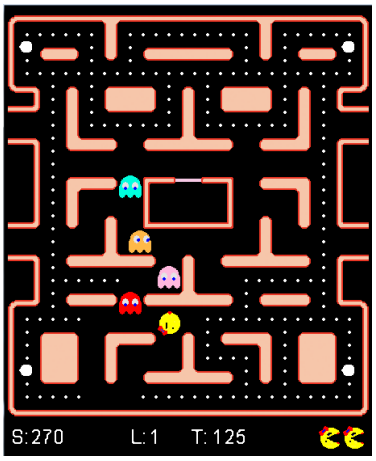

Fig. 1: A screen shot of the Ms. Pac-Man vs. Ghosts video game.

The goal of the player is to maximize the score and he obtains points each time he eats a pill (10), a power pill (50), or an edible ghost (200). Actually, the player obtains several more points if he eats more than one ghost in a row (there is a multiplier). A level or maze is completed when Pac-Man eats all the pills and power pills or he survives for 3000 turns. A turn is a *tick* of the game in which all elements perform an action and the game is updated; there are 25 ticks per second.

3 Case-Based Reinforcement Learning

3.1 Case-based Reasoning

Case-based reasoning (CBR) [1] is a learning and problem solving technique based on the intuition that similar problems usually have similar solutions. Instead of trying to solve each problem from scratch, problems are solved by looking for similar past experiences and adapting their solution to the current context. In some domains, this adaptation process can be more effective than computing a new solution from scratch because it can take advantage of available expert domain knowledge. CBR suggests then a model of reasoning that puts together problem solving, understanding and learning from the past.

A *case* represents one of these experiences and it is usually made of a description of the problem and its context, a description of the solution adopted and some measure of how successful that solution was to solve the problem. A *case base* is a collection of cases or past experiences.

Figure 2 shows the typical cycle of a CBR system. When the system faces a new problem, it looks for past similar problems in the case base (retrieval). The definition of similarity is usually domain dependent and can take advantage of the available expert knowledge. Once the most similar case has been retrieved,

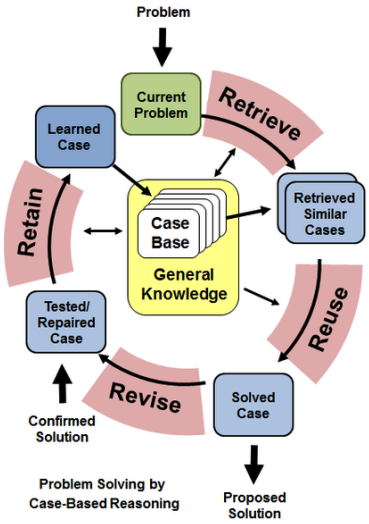

Fig. 2: CBR cycle according to Aamodt and Plaza [1].

its solution is adapted to work in the current context since the retrieved case is usually not identical to the current problem (reuse). The adaptation stage is also domain dependent. The new adapted solution is used to solve the current problem with more or less success (revise) and can be checked by a domain expert. Finally, a new case is created to describe the current problem and its solution, and the case is incorporated to the case base (retain).

The quality of a CBR system depends, therefore, on the ability to understand new situations and detect similar past experiences, the ability to adapt those past experiences to the current context, and the ability to integrate new experiences into its memory.

3.2 Reinforcement Learning

Reinforcement Learning (RL) [11] tries to solve the problem of finding sequences of actions an agent should take in order to maximize some numerical reward. The agent is not told about what actions to take, but instead it has to discover which one returns the biggest reward in the long term by trial and error.

An RL agent interacts with the environment in discrete time steps, in each of which it will receive a representation of such environment known as *state*. The agent must chose then the best action to execute in the current state. That action will lead it to a new state, and then the agent will receive a reward. The goal of the agent is to maximize total reward obtained during a simulation. Note that the optimal action might not be the one that maximizes the instant reward (a greedy approach), but other action that allows to obtain a higher total reward during the simulation.

Algorithm 1: Q-learning algorithm

```
Initialize Q(s,a) arbitrarily;
repeat
    Initialize s;
    repeat
        Choose a from s using policy derived from Q;
        Take action a, observe r, s';
        Q(s,a) ← Q(s,a) + α[r + γmax_a' Q(s',a') − Q(s,a)];
        s ← s';
    until s is terminal;
until no more episodes;
```

One of the most important challenges in RL is how to keep balance between exploitation and exploration. To get the best reward, the agent might be tempted to select actions that worked in the past. But to discover new sequences of actions it has to take decisions that it has not taken before. So the agent must try as much actions as possible and enough amount of times each, and progressively favor those that appear to be the best.

One can now identify some key elements in reinforcement learning:

- **State:** A representation of the environment at some point of the learning process.
- **Policy:** Defines the behavior of the agent at some time. It maps a given state to all possible actions the agent can take from there.
- **Reward function:** It defines what are the good and bad actions for the learner. It gives the rewards for doing an action from some state.
- **Value function:** Specifies what is good in the long run for the agent. It gives values to states depending on the expected reward the learner can reach from there in the future. It sets then which states are more desirable for the agent to be into.

One of the most important breakthroughs in reinforcement learning was the development of Q-learning algorithm [12] (see Algorithm 1). It works by handling a table (called *Q-table*) that maps each state to all possible actions the agent can perform in that state, and assigns them a numerical value, depending on how good they are for the agent in the long term. These are the values the algorithm will update each simulation cycle. The following equation states how the value of an action $\boldsymbol{a}$ taken in a state $\boldsymbol{s}$ is modified each time the agent performs it:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

where α y γ are learning and discount rates, respectively.

In this case, the learned Q-function approaches directly to its optimal values, independently on the chosen policy. This enormously simplifies the algorithm analysis and convergence proofs. The only requirement for the policy to guarantee convergence is that all state-action pairs must continue to be updated.

Algorithm 2: Case-based Q-learning algorithm

```
Initialize case base arbitrarily;
repeat
    Initialize case c;
    repeat
        Search most similar case c in the case base, get similarity s between
          them ( 0 ≤ s ≤ 1 );
        Choose a from c';
        Take action a, observe r, c";
        Q(c, a) ← Q(c, a) + α[s ∗ r + γmax_a' Q(c'', a') − Q(c, a)];
        if c and c' are distinct enough then
            add c' to case base;
        end
        c ← c'';
    until no more steps in the episode;
until no more episodes;
```

3.3 Using cases to approximate the Q function

In the general case, the number of possible states and actions leads to a such combinational explosion of pairs that it is not possible to store the Q-table in memory. Even if we could store it, we would require a huge number of simulations to update its values and converge to the optimal policy, since we might not find the same states in different simulations.

The Q-table can be generalize to a $Q : S \times A \rightarrow \mathbb{R}$ function that maps pairs of states and actions to their expected future reward during the simulation. There are different approaches to approximate the Q function using Neural Networks [7], decision trees [9] and many other machine learning techniques. In this work we propose to use a case base.

The use of cases has some interesting advantages such as we can inject expert domain knowledge in the similarity measure used to retrieved similar cases and in the adaptation stage. It has also some disadvantages compared to other techniques that build domain models because CBR is memory intensive and usually slow with a big number of cases.

Algorithm 2 shows the modified version of Q-learning adapted to work with a case base. The first difference is that now we retrieve the most similar case to the current state. Since the retrieved case will be hardly an exact match to the current state, its solution must be adapted. We also use the similarity value to weight the contribution of the reward during the update of the Q-value so that the influence will be greater for higher similarities. Finally, we only store the new state as a new case in the case base if it is different enough from the cases already known. That way, we can limit the number of cases and the memory requirements of the algorithm and tune the re-usability of the cases.

4 A Case-Based Reinforcement Learning bot for Pac-Man

In this section we describe the different decisions we made to implement our Pac-Man bot. These decisions involve the features chosen to represent the game state, the granularity of the actions learned, the similarity function used to compare similar situations, the adaptation strategy to reuse the retrieved solution in the current context, the case base management policy and the configuration of the meta-parameters for the Q-learning algorithm. Such meta-parameters will be a learning factor $\alpha = 0.2$, a discount rate $\gamma = 0.8$ and a ϵ-greedy exploration approach with an initial parameter $\epsilon = 1$ that decreases linearly with the number of episodes.

4.1 Cases

A case represents the values of the Q-function for one particular game state. Remember that $Q : S \times A \rightarrow \mathbb{R}$ is a function that maps states and actions to expected reward values in the long term. Our cases contain, therefore, the description of a particular game state and a set of (action, Q-value) pairs representing the expected rewards in the long term if those actions were chosen in the current state.

A game state is represented by the state of the gameboard: the position of the walls, pills, power pills, ghosts and Pac-Man. The number of possible states is too high so we need to work with a more abstract representation. In order to choose a set of features, we identified the most relevant information for a human player:

- Distances to the closest pill in each direction (d_1).
- Distances to the closest power pill in each direction (d_2).
- Distances to the closest non edible ghost in each direction (d_3).
- Distances to the closest edible ghost in each direction (d_4).

The maze is represented as a graph in the game engine, so these are the distances in number of nodes from one node to another one. Distances in every direction are calculated using breadth-first search. Since there are 4 directions (up, down, left, right), each d_i is a vector of 4 components and the game state is described with 16 parameters. In order to speed up the learning process, we precompute the distance between all the positions in the map and sort them by distance so that the search of the closest entity can be done in lineal time.

Even with this abstract representation of the game state, the number of possible states is huge. In order to reduce the search space, we discretized the distances in 5 possible values: *WALL* (when there is a wall in the path, so Pac-Man can't go in that direction), *VERY CLOSE* (less than 4 units, that is one *tile*), *CLOSE* (between 4 and 48 units 1-12 tiles), *MEDIUM* (between 49 and 224 units, 13-56 tiles) and *FAR* (225 units, 57 tiles, or more). This way, the

number of possible states is "only" $5^{16} = 152,587,890,625$. As a reference, the Pacman labyrinth size is 28×26 tiles.

Regarding the actions, Pac-Man can move in any of the four main directions: *UP, DOWN, RIGHT* and *LEFT*, as long as there is not a wall blocking its path. Staying still is not allowed, so there are as much as four possible actions in each state. It is important to note that decisions are only made when Pac-Man reaches an intersection, to avoid hesitation, or if there is a ghost close to Pac-Man.

4.2 Similarity function

We need a way of comparing cases, so when the agent is in some situation, it can retrieve the most suitable case from the case base. This involves searching the most similar case and then using its solution adapted to the current context. There are many ways in which cases can be compared, using different kinds of linear and non-linear functions.

In this work we use a simple linear combination of the features described in the previous section, so we can use the different weight to give more or less importance to each feature depending on the learning goal:

$$sim(c, c') = 1 - dist(c, c')$$
$$dist(c, c') = \sum_{i=1}^{4} w_i * dist_v(d_i, d'_i)$$
$$dist_v(d_i, d'_i) = \sum_{i=1}^{4} j \begin{cases} j = 1 \text{ if } d_i = d'_i \\ j = 0 \text{ if } d_i \neq d'_i \end{cases}$$

The role of the similarity function is to increase the reusability of each case so the Q-values can be used not only in that game state but also in similar ones.

4.3 Adaptation

In Pac-Man the adaptation step is very straightforward, since actions are the same for all cases. The only situation in which best action could not be applied to a case is when there is a wall blocking the path, so that best action means trying to go in a direction that is not available for the agent. When this occurs, the selected case is discarded and the second more similar one will be used instead.

4.4 Case base management

Case base management becomes a key task in the CBR process in order to maintain good performance. We use two different strategies to maintain a reasonable number of cases in the case base.

First, we only add a new case to the case base if it is different enough from the existing ones. The intuition is that two very similar cases will probably have

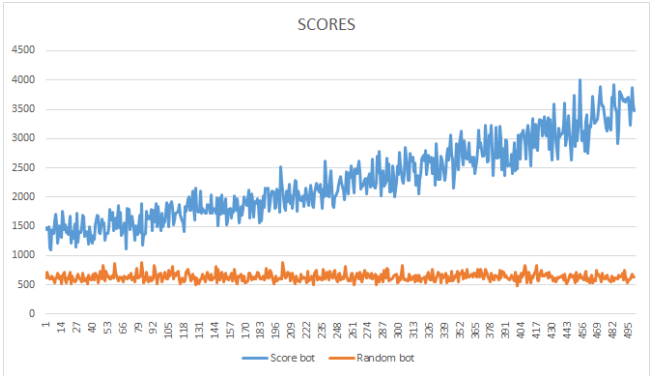

Fig. 3: Scores of the RL agent and a random agent vs. number of learning episodes.

similar Q-values associated with the actions so we do not need to remember both of them. This is made by looking at the result of the similarity function that compares both cases during the retrieval stage. The current threshold to learn a case is a similarity value that will depend on the experiment parameters.

Second, when the number of learned cases reaches 100000, we remove 10% of them. In particular, we look for pairs of similar cases in the case base with the same best action (according to the Q-values) and we "forget" the one that has been retrieved less often. The intuition is that both cases represent similar situations in which the agent should make the same decision so erasing one of them should not lead to an important loss of information.

5 Experimental evaluation

We performed three different experiments with different goals and different weights in the similarity function. The main aim is to make the bot to develop different behaviors depending on what information is more important and what type of reward the agent gets during the simulation. All the experiments consisted in playing 10000 episodes or games.

Experiment 1 In this first experiment the goal is make the bot get as much score as possible. The weights used in the similarity function were: pill (0.9), power pill (0), none edible ghost (0.05) and edible ghost (0.05). The similarity thresholds to learn new cases was 0.001. The reward matches the game score.

Figure 3 shows the evolution of the score of our RL agent (blue) and a random agent (orange) that takes random decision when it reaches an intersection. Since the most important parameters are the distances to the pill, the RL agent tries to eat them only changes when there is a ghost close. The final agent is able to reach 3510 points of average, and hits a maximum of 6150 during some of

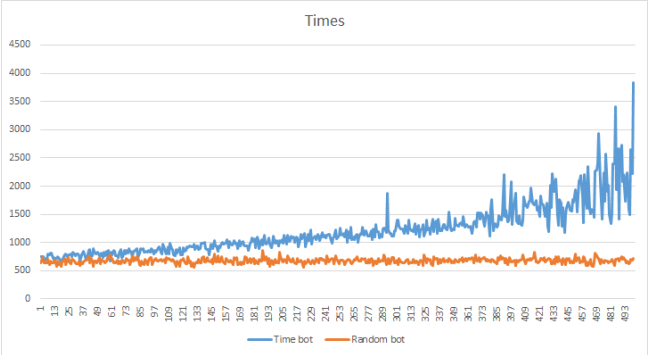

Fig. 4: Turns alive of the RL agent and a random agent vs. number of learning episodes.

the episodes at the end of the simulation when the bot is just playing and not learning any more.[2]

Experiment 2 In the second experiment the goal is to avoid ghosts, in order to survive as much time as it can. The weights used in the similarity function were: pill (0), power pill (0), non edible ghost (0.95) and edible ghost (0.05). The similarity thresholds to learn new cases was 0.01. The agent obtains a reward every turn it is alive.

Figure 4 shows the time the agents are alive vs. the number of learning episodes. The agent learns to stay alive from only 2 state parameters: the distances to the edible and non edible ghosts. Something interesting is that when Pac-Man completes a level if he survives for 3000 turns even if there are still pills in the board, so the final RL agent is able to complete 3 or 4 levels in some of the episodes.[3]

Experiment 3 The goal of the final experiment is to train an agent to obtain both high score and remain alive. The weights used in the similarity function were: pill (0.2), power pill (0.05), non edible ghost (0.7) and edible ghost (0.05). The similarity thresholds to learn new cases was 0.025. The agent obtains a reward every turn it is alive and every time it gets game points.

Figure 5 shows the evolution of the score and the number of turns the agents keep alive.[4]

Table 1 summarizes the stats of the agent performed in every experiment, when played 100 episodes without learning and against *Legacy Ghosts*. Those are different and harder ghosts than the default set used for learning.

[2] There is a video of this experiment in the following link: `https://youtu.be/-fD8t7tRjG0`

[3] `https://youtu.be/4phEjPAGrbk`

[4] `https://youtu.be/fftrYjzTLPg`

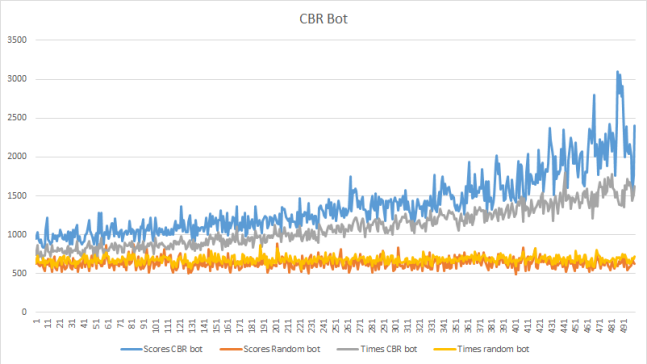

Fig. 5: Score and turns alive of the RL agent and a random agent vs. number of learning episodes in experiment 3.

Bot	Max Score	Max Time	Avg Score	Avg Time
Experiment 1	6150	1491	3510	1078
Experiment 2	6320	5653	2022	1866
Experiment 3	4310	4047	1918	1523

Table 1: Agent result for each experiment.

6 Related work and conclusions

Although the techniques described in this paper are not yet massively used in commercial games, there is previous research work that has explored similar ideas. For example, Gallagher and Ledwich [3], and later Bom and Henken [2] studied about training Pac-Man bots using neural networks, with some interesting results, starting the path of mastering this game with AI methods.

Reinforcement learning has also been used in other domains such as first person shooters. Lample and Chaplot [4] made a very successful research of this area using the FPS game *Doom*.

There is also interesting CBR research with video games. Sharma et al. [10] combined case based reasoning and reinforcement learning to train bots for playing real time strategy games.

In this paper, we have also joined together CBR and RL to study their usability in video games in general, and to research about the feasibility of building a learning agent that could develop different behaviors using these methods to beat Ms. Pac-Man game specifically.

We have shown that both techniques are well suited for learning basic, independent strategies, reaching an encouraging performance. Unfortunately, when different, and in some situations opposite, strategies must be taken into account, the results are no so promising. We plan to explore the causes of this fact and

look for ways to compensate the confronting relationship between strategies, and compare the resultant bots to related works like [8].

Whole code of this research can be found in the following repository: `https://github.com/fedomi/PacMan-vs-Ghosts.git`

References

1. Aamodt, A., Plaza, E.: Case-based reasoning: Foundational issues, methodological variations, and system approaches. AI Commun. 7(1), 39–59 (Mar 1994)
2. Bom, L., Henken, R., Wiering, M.: Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. In: ADPRL. pp. 156–163. IEEE (2013), `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6596003`
3. Gallagher, M., Ledwich, M.: Evolving pac-man players: Can we learn from raw input? In: Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games, CIG 2007, Honolulu, Hawaii, USA, 1-5 April, 2007. pp. 282–287 (2007)
4. Lample, G., Chaplot, D.S.: Playing FPS games with deep reinforcement learning. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA. pp. 2140–2146 (2017)
5. Lucas, S.M.: Ms. Pac-Man competition (2007-2011). `http://dces.essex.ac.uk/staff/sml/pacman/PacManContest.html`, accessed: 2017-06-25
6. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge, MA, USA (1998)
7. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature 518(7540), 529–533 (02 2015)
8. Pepels, T., Winands, M.H.M., Lanctot, M.: Real-time monte carlo tree search in Ms Pac-Man. IEEE Transactions on Computational Intelligence and AI in Games 6(3), 245–257 (Sep 2014)
9. Pyeatt, L.D., Howe, A.E.: Decision tree function approximation in reinforcement learning. Tech. rep., In Proceedings of the Third International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Models (1998)
10. Sharma, M., Holmes, M.P., Santamaría, J.C., Irani, A., Jr., C.L.I., Ram, A.: Transfer learning in real-time strategy games using hybrid CBR/RL. In: Veloso, M.M. (ed.) IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. pp. 1041–1046 (2007)
11. Sutton, R.S., Barto, A.G.: Introduction to Reinforcement Learning. MIT Press, Cambridge, MA, USA, 1st edn. (1998)
12. Watkins, C.J.C.H.: Learning from Delayed Rewards. Ph.D. thesis, King's College, Cambridge, UK (May 1989)
13. Williams, P.R.: Ms. Pac-Man vs Ghosts AI. `http://www.pacmanvghosts.co.uk`, accessed: 2017-06-25
14. Williams, P.R., Liebana, D.P., Lucas, S.M.: Ms. Pac-Man versus Ghost team CIG 2016 competition. In: IEEE Conference on Computational Intelligence and Games, CIG 2016, Santorini, Greece, September 20-23, 2016. pp. 1–8 (2016)