Artificial Intelligence in Adversarial Real-Time Games: Papers from the 2012 AIIDE Workshop
AAAI Technical Report WS-12-15

CLASS$_{Q-L}$: A Q-Learning Algorithm for Adversarial Real-Time Strategy Games

Ulit Jaidee, Héctor Muñoz-Avila

[1]Department of Computer Science & Engineering; Lehigh University; Bethlehem, PA 18015
ulj208@lehigh.edu | munoz@cse.lehigh.edu

Abstract

We present CLASS$_{Q-L}$ (for: class Q-learning) an application of the Q-learning reinforcement learning algorithm to play complete Wargus games. Wargus is a real-time strategy game where players control armies consisting of units of different classes (e.g., archers, knights). CLASS$_{Q-L}$ uses a single table for each class of unit so that each unit is controlled and updates its class' Q-table. This enables rapid learning as in Wargus there are many units of the same class. We present initial results of CLASS$_{Q-L}$ against a variety of opponents.

Introduction

Reinforcement learning (RL) is an unsupervised learning model in which an agent learns directly by interacting with its environment. The agent pursues to maximize some signal from the environment. This signal basically tells the agent the effects of the action's it performed (Sutton and Barto, 1998).

One of the challenges of applying RL to real-time strategy games (RTS) such as Wargus is that these games have large state and action spaces. Games are played in 2-dimensional maps. States include information about:

- The number of resources and their (x,y)-coordinates in a map
- Information about each unit, including its class (e.g., if the unit is an archer) and its (x,y)-coordinates.
- Information about each building and its (x,y)-coordinates

In a given scenario there can be dozens such units and buildings. The action space of Wargus is also very large. Units execute commands, depending on their class, including:

- To construct a building, indicating the class of building (i.e., barracks) and its (x,y)-coordinate.
- To harvest a resource, indicating the class of resource (e.g., wood) and its (x,y) coordinate.
- To move to an (x,y)-coordinate
- To attack an enemy unit or structure located in an (x,y)-coordinate.

Given the large size of the action and state space, it is very difficult to use RL algorithms to control the full scope of real-time strategy games. Existing research on using RL for these kinds of games typically focuses on some aspect of the game. We will discuss some of this research in the Related Work section.

We introduce CLASS$_{Q-L}$, an application of the RL algorithm Q-learning (Watkins, 1989) that is capable of playing the full-scope of the Wargus real-time strategy game. We reduce the size of the state-action space by having a separate Q-table for each class of unit and building and filtering useful state and action information that is customized for each class.

In the next section we provide a quick overview of the Q-learning algorithm. Then we present the CLASS$_{Q-L}$ algorithm. Afterwards, we discuss how we model Wargus in CLASS$_{Q-L}$. Then we present an empirical evaluation. Next we discuss related work and in the final section we make some concluding remarks.

Q-Learning

Q-learning is a frequently use reinforcement learning technique for two main reasons. First, it doesn't require knowing the dynamics of the environment (i.e., probability distributions over resulting states when actions are taken in a given state). Second, it can bootstrap (i.e., estimating over existing estimates) enabling it to learn faster in many situations.

Conceptually, Q-learning works by learning an action-value function that gives the expected utility of taking a given action in a given state. The estimated value of action-value function of taking action a in state s at the time t^{th} is denoted as $Q(s_t, a_t)$. Q-learning is defined by:

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

Where r_{t+1} is the reward observed, the parameter α controls the learning rate ($0 < \alpha \leq 1$), and γ is a factor discounting the rewards obtained so far ($0 \leq \gamma < 1$).

Given learned Q-values, when reaching an state s, Q-learning algorithms typically select the "greedy" action a. That is the action a that has the highest Q(s,a) value with a probability $1 - \in$ and select a random action with a probability $\in$ (with $0 < \in < 1$). Typically $\in$ is set very small (i.e., close to 0), so that most of the time the greedy action is selected. This method of choosing between the greedy action and the random action is called $\in$-greedy and it enables Q-learning to escape local maxima.

After a potentially large number of iterations, Q-values are learned such that when the greedy action is always chosen the return at time t is maximized, where the return is defined as the summation of the rewards obtained after time t and until the episode ends.

The CLASS$_{\text{Q-L}}$ Algorithm

CLASS$_{\text{Q-L}}$ is an algorithm that is developed to play complete Wargus games. In Wargus players compete against one another by raising and managing armies of different classes of units. There are various classes of units in each team and units in each class have different set of actions. The basic idea CLASS$_{\text{Q-L}}$ is to maintain a single table for all units of the same kind. The motivation for this is to speed learning by updating the Q-values for every unit's action. Here, we include building structures whose action is to build a certain type of units and peasants which can construct building structures.

```
CLASS_{Q-L}($s_0$, $\Delta$, $\mathcal{Q}$, $\mathcal{C}$, $\mathcal{A}$, $\alpha$, $\gamma$, $\varepsilon$) =
 1: $s \leftarrow s_0$; start-episode();
 2: while episode continues
 3:     wait($\Delta$)
 4:     $\mathbb{s}' \leftarrow$ GETSTATE()
 5:     for each class $C \in \mathcal{C}$
 6:         $s' \leftarrow$ GETABSTRACTSTATE($\mathbb{s}'$, $C$)
 7:         $A \leftarrow$ GETVALIDACTIONS($\mathcal{A}_C$, $s'$)
 8:         Q $\leftarrow \mathcal{Q}(C)$
 9:         for each unit $c \in C$
10:             if unit $c$ is idle
11:                 if RANDOM(1) $\geq \varepsilon$
12:                     $a \leftarrow$ ARGMAX$_{a' \in A}$(Q($s'$, $a'$))
13:                 else
14:                     $a \leftarrow$ RANDOM($A$)
15:                 EXECUTEACTION($a$)
16:                 $L_c \leftarrow$ concat($L_c$, <$s_c$, $a_c$, $s'$>)
17:                 $s_c \leftarrow s'$;  $a_c \leftarrow a$
    END-WHILE
    //After the game is over, update the q-tables
18: $r \leftarrow$ GETREWARD
19: for each class $C \in \mathcal{C}$
20:     Q $\leftarrow \mathcal{Q}(C)$
21:     for each unit $c \in C$
22:         for each <$s$, $a$, $s'$> $\in L_c$
23:             Q(s, a) $\leftarrow$ Q(s, a) + $\alpha$ [$r$ +
                    $\gamma$ARGMAX$_{a'}$ (Q($s'$, $a'$) – Q(s, a)]
24: return $\mathcal{Q}$
```

CLASS$_{\text{Q-L}}$ receives as inputs the starting state s_0, a waiting time Δ, the collection of Q-values for each individual classes $\mathcal{Q}$, the set of all classes $\mathcal{C}$, the set of all possible action $\mathcal{A}$, the step-size parameter α the discount-rate parameters γ from Q-learning, and the parameter ε for the ε-greedy selection of action.

CLASS$_{\text{Q-L}}$ works in two phases. In the first phase we use the Q-values learned in previous episodes to control the AI while playing the game. In the second phase we update the Q-values from the sequence of (s,a,s') triples that occurred in the episode that just ended.

CLASS$_{\text{Q-L}}$ initializes s to the initial state s_0 and starts the episode (Line 1). During an episode (Line 2), CLASS$_{\text{Q-L}}$ periodically waits (Line 3) and then observes the current state $\mathbb{s}'$ (Line 4). For each class C in the set of classes $\mathcal{C}$ (Line 5), create the current state s' for the class C by customizing the observed state $\mathbb{s}'$ (Line 6). The reason why we have to do this is because different classes need different kinds of information. The size of the observe state $\mathbb{s}'$ is very large and contains various kinds of information. Each class require some of the information uniquely. We will detail the kind of information extracted in the next section.

The next step is to create the set of valid actions A of class C under current situation (Line 7). We should not use $\mathcal{A}_C$ (the set of possible actions of class C) directly because some of the actions might not be applicable in the current state. For example, peasants can build farms. However, without enough resources, Wargus will ignore this command. Therefore, Line 7 prunes invalid actions. Any action randomly chosen from this set of actions is guaranteed to be a valid action. Next, retrieve the Q-table of class C from the collection of Q-tables $\mathcal{Q}$ (Line 8).

For each unit c of class C, if the unit c is idle, CLASS$_{\text{Q-L}}$ retrieves an action a from the Q-table using ε-greedy exploration (Line 9-14). Notice that the algorithm choose an action a from the set of valid actions A, not from the set of possible actions $\mathcal{A}_C$. Then, execute the action a (Line 15).

Because this is offline learning method, Line 16 saves the set of s_c (the previous state s of unit c), a_c (the previous action a of unit c) and the current state s' for the Q-learning updates in the second phase. We wait until the end of the game to update the Q-values because we have found experimentally to be more effective to use the outcome at the end of the game. This is why we have to save the list of state-action to perform the off-line update later. Line 17 updates the previous state s_c and the previous action a_c.
In the second phase, after calculating the reward r (Line 18), we use the Q-learning update on all the members in the list L_c of each individual unit c to update Q-values of each class C (Line 19-23). Finally, returns the Q-values (Line 24).

Modeling Wargus in CLASS$_{\text{Q-L}}$

We modeled the following 12 classes in CLASS$_{\text{Q-L}}$:

1. Town Hall / Keep / Castle
2. Black Smith
3. Lumber Mill
4. Church
5. Barrack
6. Knight / Paladin
7. Footman
8. Elven Archer/ Elven Ranger
9. Ballista
10. Gryphon Rider
11. Gryphon Aviary
12. Peasant-Builder

Because the behaviors of peasants when they act as builders or harvesters are so different, we separate them in two different subsets. There is only one peasant who is the builder. The rest are all harvesters for either gold or wood and, in some situations, the peasants in this group also act as repairers.

There is no stable class in the list above because stable has no action, so the Q-table is not needed by the stable class. The peasant-harvester class also do not have its own Q-table. We create simple algorithm for job assignment to harvesters to maintain the ratio of gold to wood about 2:1. There are a few other missing classes such as Mages because they don't seem to work well when controlled by using Wargus commands.

In Wargus, the sets of actions of each class are exclusive. So, we can make the state space of Q-table smaller by having individual Q-table of each class of unit. All Q-values are zero initialized.

4.1 State Representation

Each unit type has different state representation. To reduce the number of states, we genralize levels for features that have too many values. For example, the amount of gold can be any value greater than zero. In our representation we have 18 levels for gold. Level 1 means 0 gold whereas level 18 means more than 4000 gold. We used the term "level number" for such generalizations. Here are the features of the state representations for each class:

- Peasant-Builder: level number of gold, level number of wood, level number of food, number of barracks, lumber mill built?,[1] blacksmith built?, church built?, Gryphon built?, path to a gold mine?, town hall built?
- Footman, knight, paladin, archer, ballista and Gryphon rider: level number of our footmen, level number of enemy footmen, number of enemy town halls, level number of enemy peasant, level number of enemy attackable units that are stronger than our footmen, level number of enemy attackable units that are weaker than our footmen
- Town hall: level number of food, level number of peasants
- Barrack: level number of gold, level number of food, level number of footman, level number of footmen, level number of archers, level number of ballista, level number of knights/paladins
- Gryphon Aviary: level number of gold, level number of food, level number of Gryphon Rider
- Black Smith, Lumber Mill and Church: level number of gold, level number of wood

For peasants who are harvesters, we do not use any learning to choose the actions they take. Instead, we simply balance between the amount of gold and the amount of wood about two units of gold per one unit of wood. Another work of harvesters is to repair damage structures if there are some that need repairs.

4.2 Actions

Our model abstracts actions from units so that they are at higher-level than actual actions the units can take. The actual actions of units include moving to some location, attacking another unit or building, patrolling between two locations, and standing ground in one location. However, using the actual actions would lead to an explosion in the size of the Q-tables.

The list below is the list of all possible actions for each class.

- Peasant-Builder: build a farm, build a barrack, build a town hall, build a lumber mill, build a black smith, build a stable, build a church, and build a Gryphon aviary.
- Town Hall/Keep/Castle: train a peasant, Upgrade itself to keep/castle
- Black Smith: upgrade sword level 1, upgrade sword level 2, upgrade human shield level 1, upgrade human shield level 2, upgrade ballista level 1, upgrade ballista level 2

[1] The question mark signals that this is a binary feature

- Lumber Mill: upgrade arrow level 1, upgrade arrow level 2, Elven ranger training, ranger scouting, research longbow, ranger marksmanship
- Church: upgrades knights to paladins, research healing, research exorcism
- Barrack: Doing nothing, train a footman, train an Elven archer/ranger,
  train a knight/paladin, train a ballista
- Gryphon Aviary: Doing nothing, Train a Gryphon rider
- Footman, Archer, Ranger, Knight, Paladin, Ballista, Gryphon Rider: wait for attack, attack the enemy's town hall/great hall, attack all enemy's peasants, attack all enemy's units that are near to our camp, attack all enemy's units that have their range of attacking equal to one, and attack all enemy's units that have their range of attacking more than one, attack all enemy's land units, attack all enemy's air units, attack all enemy's units that are weaker (the enemy's units that have HP less than those of us), and attack all enemy's units (no matter what kind)

Empirical Evaluation

We conducted initial experiments for CLASS$_{Q-L}$ on a small Wargus map.

5.1 Experimental Setup

At the first turn of each game, both teams start with only one peasant/peon, one town hall/great hall, and a gold mine near them. We have five adversaries: land-attack, SR, KR, SC1 and SC2 for training and testing our algorithm. These adversaries come with the Warcraft distribution and have been used in machine learning experiments before (see Related Work section).

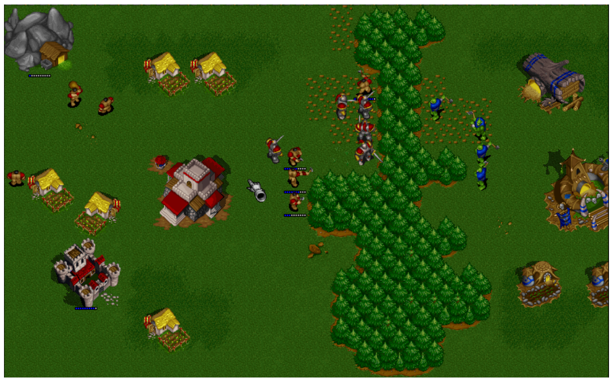


Figure 1: The screen capture of the small map from Wargus game.

These adversaries can construct any type of unit unless the strategy followed discards it (e.g., land-attack will only construct land units. So units such as gryphons are not built):

- Land-Attack: This strategy tries to balance between offensive/defensive actions and research. It builds only land units.
- Soldier's Rush (SR): This attempts to overwhelm the opponent with cheap military units in an early state of the game.
- Knight's Rush (KR): This attempts to quickly advance technologically, launching large offences as soon as knights are available. Knoghts is the strongest unit in the game
- Student Scripts (SC1 & SC2): These strategies are the top two competitors created by students for the tournament in a classroom.

We trained and tested CLASS$_{Q-L}$ by using **leave-one-out-training** as the model of our experiment processes. We remove from the training set the adversary that we want to compete against. For example, if we want to experiment CLASS$_{Q-L}$ versus SC1, the set of adversaries that we use for training is {land-attack, SR, KR, SC2}.

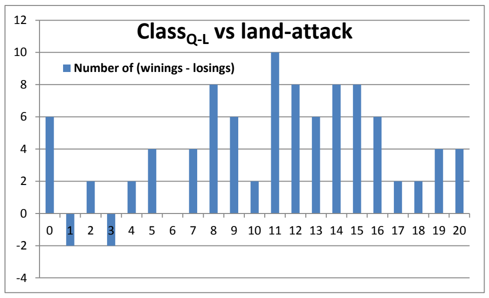
All experiments were performed on the 32 x 32 tile map shown in Figure 1. This is considered an small map in Wargus. Each competitor starts in one side of the forest that divides the map into two parts. We added this forest to give time to opponents to build their armies. Otherwise, CLASS$_{Q-L}$ was learning a very efficient soldier rush and defeating all opponents including SR very early in the game.

5.2 Results

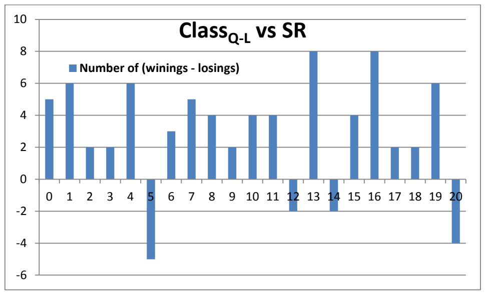
Our performance metric is: (number of wins – number of loses) with m training rounds. First we match CLASS$_{Q-L}$ against each opponent with no training (m = 0). Then we play against each opponent after one round of training using leave-one-out training (m = 1). We repeat this until m = 20. We repeat each match 10 times and compute the average metric. So the total number of games played in this experiment is 21*10 = 210 games. Including training, the total number of games run in this experiment is 210 * 3 = 630 games.

Our performance metric provides a better metric than the difference in Wargus score (our score – opponent's score) of the game because the lower score difference can mean a better performance than a larger score difference. This is due to how the Wargus score is computed. For example, our team can win the opponent very fast and the score we got is just 1735 and the game is over while the opponent got the score of 235 before the game end. In this case, the average score of (our team - opponent team) is just 1500. In another case, our team can win the opponent with the score of 3450, but the game takes very long time to run until the game is over; while the opponent team got the score of 1250. In this case, the average score of (our team – opponent team) is 2200, but it does not mean the performance is better. In fact, the performance should be worse than the previous case because it takes longer time to win.

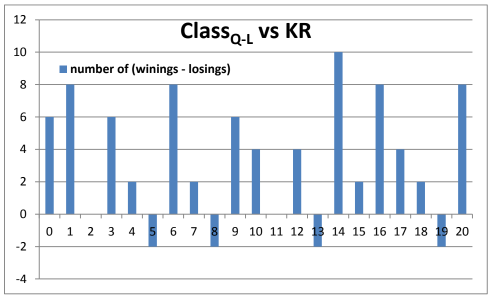
Overall the performance of $\text{Class}_{\text{Q-L}}$ is better than that of the adversaries. The x-axis shows the results after x number of iterations training in the leave-one-out setting. The first bar is x = 0 and the last bar is x = 20.



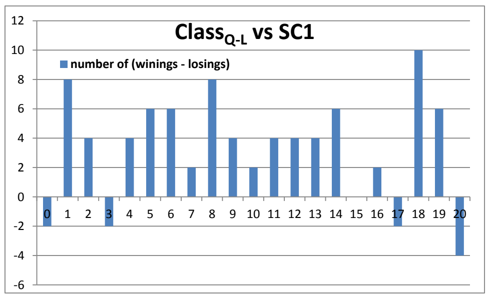
(a)



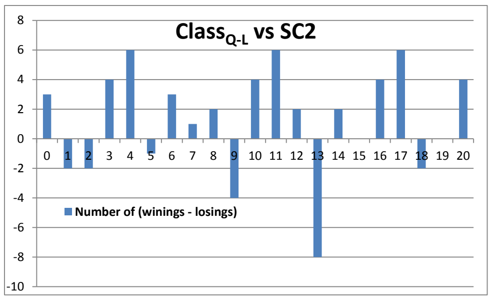
(b)



(c)



(d)



(e)

Figure 2: The results of the experiments from Wargus game: (a) $\text{Class}_{\text{Q-L}}$ vs. Land-Attack, (b) $\text{Class}_{\text{Q-L}}$ vs. SR, (c) $\text{Class}_{\text{Q-L}}$ vs. KR, (d) $\text{Class}_{\text{Q-L}}$ vs. SC1 and (e) $\text{Class}_{\text{Q-L}}$ vs. SC2.

Related Work

Wargus is an open source game that was built by reverse engineering the commercial game Warcraft 2. Wargus has been used frequently as a testbed for research purposes because it is an accurate model of real time strategy games including elements defining the genre such as research trees, players taking action in real-time, players controlling different kinds of units, gathering resources, and constructing building structures to create new units or to upgrade existing units (Mehta et al., 2009).

Our work is related to micro-management in RTS games (e.g., (Scott, 2002; Perez, 2011; Synnaeve & Bessière, 2011)). In micro-management the complex problem of playing an RTS game is divided into tasks. These tasks are accomplished by specialized components or agents. For example, an agent might be in charge of resource gathering tasks, another one of combat and so forth. In $\text{CLASS}_{\text{Q-L}}$ we take this idea further by dividing the load between classes.

Researchers have explored combining RL with techniques such as case-based reasoning to address the problem of replaying in similar scenarios for RTS games (Sharma et al., 2007). $\text{CLASS}_{\text{Q-L}}$ focuses on using reinforcement learning to play all aspects of an RTS game.

We now discuss some works using reinforcement learning in Wargus.

Ponsen et al. (2006) uses a technique called dynamic scripting (Spronck, 2006) to control all aspects of Wargus to play complete games. An script is a sequence of gaming actions specifically targeted towards a game such as in this case Wargus. In dynamic scripting these scripts are generated automatically based on feedback while playing the game. Ponsen et al. (2006) combines reinforcement learning and evolutionary computation techniques to evolve scripts. Like our work in $\text{CLASS}_{\text{Q-L}}$ the state information is heavily engineered. Their work also reports good results versus the SR and KR as in our case but, unlike CLASS$_{Q-L}$, their system couldn't overcome the land attack.

Rørmark (2009) uses the idea of micro-management described before to have specialized experts on a number of tasks in Wargus games. Similar to dynamic scripting it uses RL to learn strategic-level decisions.

Jaidee et al. (2012) reports on the use of goal-driven autonomy in Wargus. Goal-driven autonomy is introspective models were an agent examines the effects of its own actions to adjust its behavior over time. Jaidee et al. (2012) uses reinforcement learning to reason about long-term goals. Unlike our work, Jaidee et al. (2012) reports only combat tasks in Wargus. Combat units are fixed at the beginning of the game and building structures are not available.

Marthi et al. (2005) uses concurrent ALISP in Wargus games. The basic premise of that work is the user specifying a high-level LISP program to accomplish Wargus tasks and reinforcement learning is used to tune the parameters of the program. This work was demonstrated for resource gathering tasks in Wargus.

Conclusions

We presented CLASS$_{Q-L}$ a Q-learning algorithm to play complete Wargus games. CLASS$_{Q-L}$ uses a (1) carefully engineered state and action space information and (2) learns Q-values for classes of units as opposed to single units. The latter enables the rapid learning of the units' effective control because each time a unit performs an action, its class' Q-values are updated.

We performed initial experiments on a small map. In these experiments CLASS$_{Q-L}$ is able to beat all opponents most of the time. Some of these opponents were difficult to tackle for other learning algorithms. In the near future we want to test our system in Wargus maps of larger size.

Acknowledgements

This work was supported in part by the National Science Foundation grant 0642882 and the Naval Research Laboratory.

References

Jaidee, U., Munoz-Avila, H., Aha, D.W. (2012) Learning and Reusing Goal-Specific Policies for Goal-Driven Autonomy. Proceedings of the 20th International Conference on Case Based Reasoning (ICCBR 2012). Springer.

Marthi, B., Russell, S., Latham, D., and Guestrin, C. (2005) Concurrent hierarchical reinforcement learning. In Proceedings of the 20th national conference on Artificial intelligence (AAAI-05), AAAI Press 1652-1653.

Mehta, M. and Ontañón, S. and Ram, A (2009) Using Meta-Reasoning to Improve the Performance of Case-Based Planning, in *International Conference on Case-Based Reasoning (ICCBR 2009)*, LNAI 5650, pp 210 – 224

Perez, A. U. (2011) *Multi-Reactive Planning for Real-Time Strategy Games*. MS Thesis. Universitat Autònoma de Barcelona.

Ponsen, M., Munoz-Avila, H., Spronk, P., Aha, D. (2006) Automatically generating game tactics with evolutionary learning. *AI Magazine*. AAAI Press.

Rørmark, R. (2009) Thanatos - A learning RTS game AI. MS Thesis, University of Oslo.

Scott, B. (2002) *Architecting an RTS AI*. AI game Programming Wisdom. Charles River Media.

Sharma, M., Holmes, M., Santamaria, J., Irani, A., Isbell, C., and Ram, A (2007) Transfer learning in real-time strategy games using hybrid CBR/RL. In Proceedings of the 20th international joint conference on Artifical intelligence (IJCAI-07). Morgan Kaufmann Publishers Inc.

Spronck, P. (2006). Dynamic Scripting. *AI Game Programming Wisdom 3 (ed. Steve Rabin)*, pp. 661-675. Charles River Media, Hingham, MA.

Sutton, R.S., & Barto, A.G. (1998). *Reinforcement learning: An introduction*. MIT Press, Cambridge, MA.

Synnaeve, G., Bessière, P. (2011) A Bayesian Model for RTS Units Control applied to StarCraft. CIG (IEEE).

Watkins, C.J.C.H., (1989), *Learning from Delayed Rewards*. Ph.D. thesis, Cambridge University