PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
GRADUATE PROGRAM IN COMPUTER SCIENCE

META-LEVEL REASONING IN REINFORCEMENT LEARNING

JIÉVERSON MAISSIAT

Thesis presented as partial requirement for obtaining the degree of Master in Computer Science at Pontifical Catholic University of Rio Grande do Sul.

Advisor: Prof. Felipe Meneguzzi

Porto Alegre
2014

REPLACE THIS PAGE
WITH THE
PRESENTATION TERM

To Caroline Seligman Froehlich

“There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable.There is another theory which states that this has already happened.”
(Douglas Adams)

ACKNOWLEDGMENTS

I would like to thank people who contributed, not just to this work, but also to the good time spent during my masters degree.

First of all, to my advisor Felipe Meneguzzi for all support, help, insight and ideas, making it possible to end this work the way it is.

To prof. Rafael Bordini for the valuable time dedicated during the course of this work.

To prof. Avelino Francisco Zorzo for the encouragement to discover my own area of interest.

To anonymous reviewers of SBGames 2013 whose feedback our paper allowing me to refine this work.

To BTHAI and BWAPI developers for creating and documenting the tools that have made this work possible.

To Caroline Seligman Froehlich for showing me that it is always worth giving up what makes you unhappy to run after your dreams.

To my parents Fernando Maissiat and Cerli Maissiat and my brother Jackson for providing me full support and good family structure.

META-LEVEL REASONING IN REINFORCEMENT LEARNING

ABSTRACT

Reinforcement learning (RL) is a technique to compute an optimal policy in stochastic settings where actions from an initial policy are simulated (or directly executed) and the value of a state is updated based on the immediate rewards obtained as the policy is executed. Existing efforts model opponents in competitive games as elements of a stochastic environment and use RL to learn policies against such opponents. In this setting, the rate of change for state values monotonically decreases over time, as learning converges. Although this modeling assumes that the opponent strategy is static over time, such an assumption is too strong with human opponents. Consequently, in this work, we develop a meta-level RL mechanism that detects when an opponent changes strategy and allows the state-values to "deconverge" in order to learn how to play against a different strategy. We validate this approach empirically for high-level strategy selection in the *Starcraft: Brood War* game.

Keywords: artificial intelligence, learning, reinforcement learning, high level strategy, starcraft, games.

META-LEVEL REASONING IN REINFORCEMENT LEARNING

RESUMO

Reinforcement learning (RL) é uma técnica para encontrar uma política ótima em ambientes estocásticos onde, as ações de uma política inicial são simuladas (ou executadas diretamente) e o valor de um estado é atualizado com base nas recompensas obtida imediatamente após a execução de cada ação. Existem trabalhos que modelam adversários em jogos competitivos em ambientes estocásticos e usam RL para aprender políticas contra esses adversários. Neste cenário, a taxa de mudança de valores do estado monotonicamente diminui ao longo do tempo, de acordo com a convergencia do aprendizado. Embora este modelo pressupõe que a estratégia do adversário é estática ao longo do tempo, tal suposição é muito forte com adversários humanos. Conseqüentemente, neste trabalho, é desenvolvido um mecanismo de meta-level RL que detecta quando um oponente muda de estratégia e permite que taxa de aprendizado almente, a fim de aprender a jogar contra uma estratégia diferente. Esta abordagem é validada de forma empírica, utilizando seleção de estratégias de alto nível no jogo *Starcraft: Brood War*.

Palavras-Chave: inteligência artificial, aprendizado, reinforcement learning, high-level strategy, starcraft, jogos.

LIST OF FIGURES

LIST OF ALGORITHMS

CONTENTS

1. INTRODUCTION

For thousands of years, humans have tried to understand their own thinking and reasoning. It is difficult to understand how we are able to perceive, understand, reason, act, and, finally, learn. Artificial Intelligence (AI) is trying to understand this as well as to build agents with these capabilities. In computer games we have basically the same difficulties. It is dificult to make an agent play a game from the point of view of a regular player, with promising results [Tay11, ML10]. When humans play against other humans, it is usually difficult to forecast each individual player's actions and strategies. However, when playing against a computer, humans often start to understand the pre-programmed strategies after a few matches. Solving this problem is interesting because it can make games more dynamic and less predictable.

Reinforcement learning is a technique often used to generate an optimal (or near-optimal) agent in a stochastic environment in the absence of knowledge about the reward function of this environment and the transition function [KLM96]. A number of algorithms and strategies for reinforcement learning have been proposed in the literature [SSK05, GHG04], which have shown to be effective at learning policies in such environments. Some of these algorithms have been applied to the problem of playing computer games from the point of view of a regular player with promising results [Tay11, ML10]. However, traditional reinforcement learning often assumes that the environment remains static throughout the learning process while the learning algorithm converges. Under the assumption that the environment remains static over time, when the learning algorithm converges, the optimal policy has been computed, and no more learning is necessary. Therefore, a key element of RL algorithms in static environments is a learning-rate — parameter that defines the intensity of the learning process — that is expected to decrease monotonically until the learning converges. However, this assumption is clearly too strong when part of the environment being modeled includes an opponent player that can adapt its strategy over time.

In this work, we apply meta-level reasoning [CR08, UJG08] to reinforcement learning [SD03] and allow an agent to react to changes of strategy by the opponent. Our technique relies on using another reinforcement learning component to vary the learning-rate as negative rewards are obtained after the policy converges, allowing our player agent to deal with changes in the environment induced by changing strategies of competing players. To collect results, we set up an environment for tests in the well known game StarCraft: Brood War. The first part of this work was submitted and is already published at SBGames 2013 [MM13]. Our experiments have shown that by using our proposed meta-level reasoning, the agent start dealing with environment changes earlier than with already known methods.

This work is organized as follows: in Chapter 2 we review the main concepts used for this work: the different kinds of environments (2.1), some concepts of machine learning (2.3) and reinforcement learning (2.4), and an explanation of the StarCraft game domain (2.7); in Chapter 3 we describe our solution. Finally, we demonstrate the effectiveness of our algorithms through empirical experiments and results in Chapter 4.

2. BACKGROUND

2.1 Environments

In the context of multi-agent systems, the environment is the world in which agents act. The design of an agent-based system must take into consideration the environment in which the agents are expected to act, since it determines which AI techniques are needed for the resulting agents to accomplish their design goals. Environments are often classified according to the following attributes [RN09]: observability, determinism, dynamicity, discreteness, and the number of agents.

The first way to classify an environment is related to its observability. An environment can be unobservable, partially observable, or fully observable. For example, the real world is partially observable, since each person can only perceive what is around his or herself, and usually only artificial environments are fully observable. The second way to classify an environment is about its determinism. In general, an environment can be classified as stochastic or deterministic. In deterministic environments, an agent that performs an action a in a state s always result in a transition to the same state s', no matter how many times the process is repeated, whereas in stochastic environments there can be multiple possible resulting states s', each of which has a specific transition probability. The third way to classify an environment is about its dynamics. Static environments do not change their transition dynamics over time, while dynamic environments may change their transition function over time. Moreover an environment can be classified as continuous or discrete. Discrete environments have a countable number of possible states, while continuous environments have an infinite number of states. A good example of discrete environment is a chessboard, while a good example of continuous environment is a real-world football pitch. Finally, environments are classified by the number of agents acting concurrently, as either single-agent or multi-agent. In single-agent environments, the agent operates by itself in the system (no other agent modifies the environment concurrently) while in multi-agent environments agents can act simultaneously, competing or cooperating with each other. A crossword game is a single-agent environment whereas a chess game is a multi-agent environment, where two agents take turns acting in a competitive setting.

2.2 Markov Decision Process

Agents need to make careful decisions, since today's decisions can impact on tomorrow's and tomorrow's on the next day. Markov Decision Process (MDP), also known as stochastic control problems [Put94], is a model used to represent sequential decision making when its consequences are not deterministic. That is, MDP is a mathematical model used for modeling decision making where choices of actions are partially random and partially under the control of an agent. An MDP is represented by a tuple $\langle S, A, T, R\rangle$ [GHG04, Tay11] consisting of:

1. a *state space S* is a finite set of states $s \in S$. We could say that the state space is a set of bindings of values to variables. In computer games, one state represents a set of characteristics about the game, such as: the character is on the ground, is jumping, is facing a wall, is carrying certain items, the current life (health bar) from the character, etc.;

2. a *action space A* with a finite set of actions $a \in A$. Actions are responsible for change the game states. In computer games, action could be things like: jump, shoot, move left, move forward, pick up an item, etc..;

3. A *transition probabilities T* represents a set of probabilities of transitions given by $T : S \times A \mapsto S$. The probability function is generally defined as: $P(s' \mid s, a)$ and represents the transition probability of a state s to s' when an action is performed by the agent;

4. a *reward function* $R : S \times A \times S' \mapsto \mathbb{R}$. The reward is one of the key elements in the learning process, since it describes the *feedback* the agent uses to improve itself. In fighting games [GHG04], for example, the reward could be given by the difference from the values of the life bar between both characters, after an action is performed;

Figure 2.1 represents a MDP with three states (S_0, S_1 and S_2) and two actions (a_0 and a_1). In this example, each state has two avaliable actions, each of these actions lead to a new state. An action can lead to different states with different probabilities. For example: at state S_2, if the action a_0 is executed, there is 40% (0.4) chance to reaching the state S_0 and 60% (0.6) chance of remaining at the same state (S_2). After the state transition occurs through the execution of an action, the agent receives feedback that can be either positive or negative — in Figure 2.1 feedback from performing action a_0 at state S_1 and reaching state S_0 is $+5$, while performing the action a_1 at state S_2 and reaching state S_0 is -1.

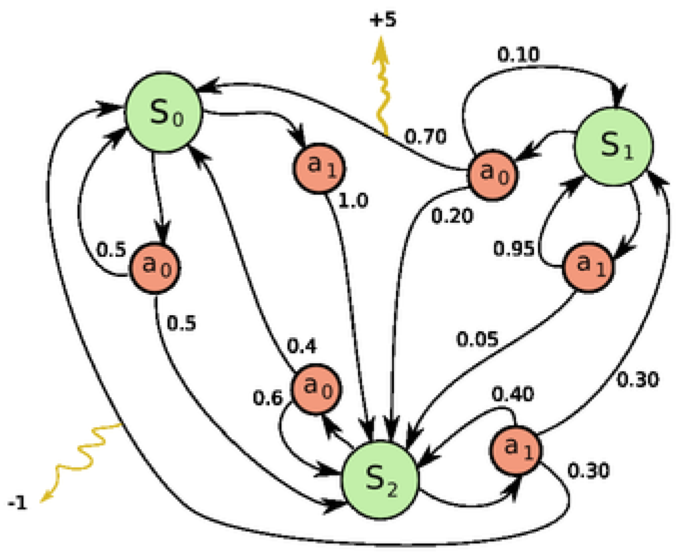


Figure 2.1 – An example of MDP with three states and two actions.

An MDP is a mapping from states to actions that tells an agent what to do at each state [Bel57, Dre02]. MDPs can be used to find optimal policies, i.e., a kind a mapping of each state to the action that yields the highest long-term reward. An optimal policy [Mor82, Mit64] is the policy that maximizes the reward obtained for each state/action [GRO03]. The Bellman's Principle of Optimality states that all parts of an optimal solution must also be optimal [Mor82]: *"An optimal policy has the property that, whatever the state and the initial decision, upcoming decisions from the resulting state must also be an optimal policy"* [Bel57, p. 83].

The Bellman equation describes the optimal policy for an MDP whose parameters are known, i.e., all states, actions, and transitions probabilities are known and are represented by Equation 2.1, where:

$$V(s) \leftarrow \left[max_a \gamma \sum_{s'}^{n} P(s'|s,a) * V(s') \right] + R(s) \tag{2.1}$$

- s represents a MDP's state;
- a represents an action, responsible for the transition between the states of the MDP;
- n represents the number of possible actions at the state s;
- s' represents the state resulting from the execution of the action a at state s;
- $V(s)$ is the long term value of s;
- $R(s)$ is the immediate reward at state s;
- γ is the *discount factor*, which determines the importance of future rewards — a factor of 0 makes the agent opportunistic [SD03] by considering only the current reward, while a factor of 1 makes the agent consider future rewards, seeking to increase their long-term rewards;

$max_a \gamma \sum_{s'}^{N} P(s'|s,a) * V(s')$ represents the best action to perform at the state s. It is part of the value iteration algorithm responsible for seeking the optimal action, at the state with the highest obtainable value from the current state.

2.3 Machine Learning

An agent is said to be learning if it improves its performance after observing the world around it [RN09]. Common issues in the use of learning in computer games include questions such as whether to use learning at all, or wether or not insert improvement directly into the agent code if it is possible to improve the performance of an agent. Russell and Norvig [RN09] state that it is not always possible or desirable, to directly code improvements into an agent's behavior for a number of reasons. First, in most environments, it is difficult to enumerate all situations an agent may find itself in. Furthermore, in dynamic environments, it is often impossible to predict all the changes over time. And finally, the programmer often has no idea of an algorithmic solution to the problem, so it is better to let a learning algorithm achieve the desired results.

Thus, in order to create computer programs that change behavior with experience, learning algorithms are employed. There are three main methods of learning, depending on the feedback available to the agent. In *supervised learning*, the agent approximates a function of input/output from observed examples. In *unsupervised learning*, the agent learns patterns of information without knowledge of the expected classification. In *reinforcement learning*, the agent learns optimal behavior by acting on the environment and observing/experiencing rewards and punishments for its actions. In this work, we focus in reinforcement learning techniques.

2.4 Reinforcement Learning

When an agent carries out an unknown task for the first time, it does not know exactly whether it is making good or bad decisions. Over time, the agent makes a mixture of optimal, near optimal, or completely suboptimal decisions. By making these decisions and analyzing the results of each action, it can learn the best actions at each state in the environment, and eventually discover what the best action for each state is.

Reinforcement learning (RL) is a learning technique for agents acting in a stochastic, dynamic and partially observable environments, observing the reached states and the received rewards at each step [SB98]. Figure 2.2 illustrates the basic process of reinforcement learning, where the agent performs actions, and learns from their feedback. An RL agent is assumed to select actions following a mapping of each possible environment state to an action. This mapping of states to actions is called a *policy*, and reinforcement learning algorithms aim to find the *optimal policy* for an agent, that is, a policy that ensures long term optimal rewards for each state.

RL techniques are divided into two types, depending on whether the agent changes acts on the knowledge gained during policy execution [RN09]. In *passive RL*, the agent simply executes a policy using the rewards obtained to update the *value* (long term reward) of each state, whereas in *active RL*, the agent uses the new values to change its policy after a certain number of iterations. A passive agent has a fixed policy: at state s, the agent always performs the same action a. Its mission is to learn how good its policy is — to learn the utility of it. An active agent has to decide what actions to take in each state: it uses the information obtained by reinforcement learning to improve its policy. By changing its policy in response to learned values, an RL agent might start exploring different parts of the environment. Nevertheless, the initial policy still biases the agent to visit certain parts of the environment [RN09], so an agent needs to have a policy to balance the use of recently acquired knowledge about visited states with the exploration of unknown states in order to approximate the optimal utility values [Gho04].

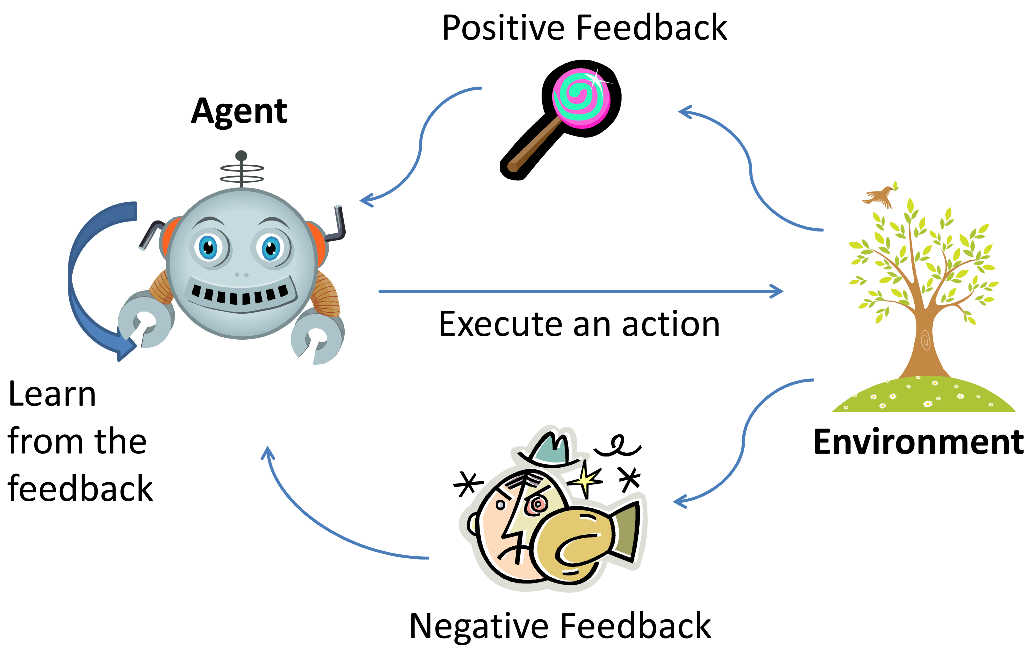


Figure 2.2 – Model to describe the process of reinforcement learning.

2.4.1 Q-Learning

Depending on the assumptions about the agent knowledge prior to learning, different algorithms can be used. When the rewards and the transitions are unknown, one of the most popular reinforcement learning techniques is *Q-learning*. This method updates the value of a pair of state and action — named state-action pair, $Q(s,a)$ — after each action performed using the immediately reward. When an action a is taken at a state s, the value of state-action pair, or Q-value, is updated using the adjustment function [AS10] shown in Equation 2.2, where:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a' \in A(s')} Q(s',a') - Q(s,a)] \tag{2.2}$$

- s represents the current state of the world;
- a represents the last action chosen by the agent;
- $Q(s,a)$ represents the value obtained the last time action a was executed at state s. This value is often called Q-value.
- r represents the reward obtained after performing action a in state s;
- s' represents the state reached after performing action a in state s;
- $a' \in A(s')$ represents a possible action from state s';
- $max_{a' \in A(s')} Q(s',a')$ represents the maximum Q-value that can be obtained from the state s', independently of the action chosen;

- α is the learning-rate, which determines the weight of new information over what the agent already knows — a factor of 0 prevents the agent from learning anything (by keeping the Q-value identical to its previous value) whereas a factor of 1 makes the agent consider all newly obtained information;
- γ is the same *discount factor* as in Section 2.2;

Once the Q-values are computed, an agent can extract the best policy known so far ($\pi^{\approx}$) by selecting the actions that yield the highest expected rewards using the Equation 2.3:

$$\pi^{\approx}(s) = \arg\max_{a} Q(s, a) \tag{2.3}$$

The complete algorithm for an exploratory Q-learning agent is show in 2.1.

Algorithm 2.1 – An exploratory Q-learning agent that returns an action after receiving immediate feedback

```
Require: $s, a, r$
1: if TERMINAL?(s) then
2:     $Q(s, None) \leftarrow r$
3:     $a' \leftarrow None$
4: else
5:     $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a' \in A(s')} Q(s', a') - Q(s,a)]$
6:     $a' \leftarrow \pi^{\approx}(s)$
7: end if
8: return $a'$
```

Q-learning converges to an optimal policy after visiting each state-action pair a sufficient number of times, but it often requires many learning episodes [WD92]. In dynamic environments, Q-learning does not guarantee convergence to the optimal policy since it is possible that the required number of leaning episodes never occurs. This occurs because the environment is always changing and demanding that the agent adapts to new transition and reward functions. However, Q-learning has been proven efficient in stochastic environments, even without guaranteed convergence [SC96, TK02, AS10]. In multi-agent systems, where the learning agent models the behavior of all other agents as a stochastic environment (an MDP), Q-learning provides the optimal solution when these other agents — or players in the case of human agents in computer games — do not change their policy choice.

2.4.2 SARSA

Q-learning is a learning algorithm that separates the policy being evaluated from the policy used to control. Sarsa — its name comes from its parameters: s, a, r, s', a' — is an alternative to the Q-learning [GHG04, SSK05] working policy being evaluated together with the policy used to control. Sarsa is based on Equation 2.4, where $0 \leq \alpha \leq 1$ represents the learning-rate and $0 \leq \gamma \leq 1$ represents the discount factor for future rewards. The adjustment of the state-action pair $(s, a) \in SxA$ is to add a minor fix (depending on α) to the old value. The fix is the immediate reward r increased by future (discounted) value of state-action $\gamma Q(s', a')$.

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma Q(s',a')] \tag{2.4}$$

The full SARSA algorithm is presented in Algorithm 2.2:

Algorithm 2.2 – SARSA

```
1: initialize $Q(s,a)$ arbitrarily
2: **for** each episode **do**
3:     initialize $s$
4:     $a \leftarrow chooseByQValue(s)$
5:     **for** each step of episode **do**
6:         $r \leftarrow getReward(s, s')$
7:         $a' \leftarrow chooseByQValue(s')$
8:         $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma Q(s',a')]$
9:         $s \leftarrow s'$
10:        $a \leftarrow a'$
11:    **end for**
12: **end for**
```

The update function is performed after each transition from a non-terminal state s (in other words, immediately after the execution of an action which does not reach the final state). If the state reached s' is terminal, then $Q(s', a')$ is set to zero [SB98]. The update function uses all elements of the tuple $\langle s, a, r, s', a' \rangle$, which represents a transition from current state-action pair to the next.

There are cases where it may become very difficult to determine the set of actions available $A(s)$ for calculate $\gamma max_{a' \in A(s')} Q(s', a')$. In these cases, it is a good idea to choose SARSA, since this requires no knowledge of $A(s)$ [GHG04].

2.4.3 Dyna-Q

Learning a model means to learn the transition probabilities, the reward values for each state and action, or both simultaneously. It is possible to find optimal policies when the parameters of the model are known. One can learn the parameters of a model through the simulation of a template. This model can be evaluated by using passive reinforcement learning, as seen previously (Section 2.4). Dyna-Q [Sut91] is a model-based approach that first builds a correct model, then finds the policies through this model [KLM96]. This approach first learns the parameters of a model, and then uses this definition of parameters to calculate the optimal policy learned about the model [AS10].

Dyna-Q can use the model to generate learning experiences — simulate the learning within a built model — and learn the Q-values more quickly than simply applying Q-learning. The agent learns both the model and the Q-values through executing its actions on the environment. The model is used to simulate the environment and update the Q-values. The better the representation of model parameters, the faster the convergence of learning will occur. The algorithm Dyna-Q operates by learning the model at each step, and then applying the technique of Q-learning to learn the model's policy.

Algorithm 2.3 – Dyna-Q

```
Require: $Q, r, s, a, numIter$

1: $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma max_{a' \in A(s')} Q(s',a') - Q(s,a))$
2: $P(s'|s,a) \leftarrow updatePAverage(s,a,s')$
3: $R(s,a) \leftarrow updateRAverage(s,a)$

4: for $i = 0$ to $numIter$ do
5:     $s' \leftarrow randomPreviouslySeenS()$
6:     $a' \leftarrow randomPreviouslyTakenA(s')$
7:     $s'' \leftarrow sampleFromModel(s',a')$
8:     $r' \leftarrow fromModel(s',a')$
9:     $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a' \in A(s')} Q(s',a') - Q(s,a)]$
10: end for

11: return $Q$
```

Dyna-Q is shown in Algorithm 2.3. The first step of Dyna-Q is to update the value of the $Q-$values using Q-learning. The transition probability ($updatePAverage$ function from the algorithm, present in Equation 2.5) is given by the number of times the agent has chosen action a at state s and reached state s', divided by the number of times the agent was in state s and has chosen the action a:

$$P(s'|s,a) = \frac{count(s,a,s')}{count(s,a)} \tag{2.5}$$

The *reward value*, is the average reward received ($updateRAverage$ function from the algorithm, present in Equation 2.6) after choosing the action a and reaching the state s:

$$R(s,a) = \frac{count(s,a) * R(s,a) + r}{count(s,a) + 1} \tag{2.6}$$

The loop *for* is responsible for generating the model. The model simulates the specified number of iterations and the Q-values are updated according to its model. First it chooses a state s' that was previously visited ($randomPreviouslySeenS$ function). After it chooses an action a' that has already been performed on state s' ($randomPreviouslyTakenA$ function). Based on this model, a new state s'' is simulated ($sampleFromModel$ function). Finally, the reward for executing action a' at state s' is given by $fromModel$ function of the algorithm, and these values are used to update the appropriate Q-value ($Q(s', a')$). For this, the algorithm makes use the Q-learning equation (Equation 2.2) as a way to learn the model's policy.

2.4.4 Prioritized Sweeping

When dealing with non-stationary environments, model-free RL approaches need to continuously relearn everything, since that the policy calculated for a given environment is no longer valid after dynamics change. This requires a readjustment phase, which drops the performance of learning and also forces the algotithm to relearn policies even for previously experienced environment dynamics [dOBdS$^+$06]. Beyond Dyna-Q, there are other model-based alternatives to deal with learning. Prioritized Sweeping [MA93] is a model-based alternative that works similar to Dyna-Q. It updates more than one state value per iteration making direct use of state values, instead of Q-values.

Prioritized Sweeping is designed to perform the same task as Gauss-Seidel iteration while using careful bookkeeping to concentrate all computational effort on the most interesting parts of the system. To determine the states that should have its values updated after each iteration, Priorized Sweeping make use of a priority queue, where the states priorities are stored and initialized as 0. Each state maintains its predecessors: all states with at least one non-zero transition probability to it under any action. After each environment observation, the transition probabilities are updated. Then state i is promoted to the top of the priority queue. Next, we continue to process further states from the top of the queue. Each state that is processed may result in the addition or promotion of its predecessors within the queue. This loop continues for a preset number of processing steps or until the queue empties. Prioritized Sweeping uses the queue to determine which states are likely to be more interesting. At each iteration, the transition probabilities T and the reward function R are estimated. The transition probabilities T can be usually updated by calculating the maximum likelihood probability [dOBdS$^+$06]. Instead of storing the transition probabilities in the form $T(s, a, s')$, Prioritized Sweeping stores the number of times the transition (s, a, s') occurred and the total number of times the situation (s, a) has been reached.

2.5 Exploration Policy

So far, we have considered active RL agents that simply use the knowledge obtained so far to compute an optimal policy. However, as we saw before, the initial policy biases learning towards the parts of the state-space through which an agent eventually explores, possibly leading the learning algorithm to converge on a policy that is optimal for the states visited so far, but not overall optimal (a local maximum). Therefore, active RL algorithms must include some mechanism to allow an agent to choose different actions from those computed with incomplete knowledge of the state-space. Such a mechanism must seek to balance exploration of unknown states and exploitation of the currently available knowledge, allowing the agent both to take advantage of actions he knows are optimal, and exploring new actions [AS10].

In this work we use an exploration mechanism known as ϵ-greedy [RGK09]. This mechanism has a probability ϵ to select a random action, and a probability $1 - \epsilon$ to select the optimal action learned so far (one that has the highest Q-value). In order to make this selection we define a probability vector over the action set of the agent for each state, and use this probability vector to bias the choice of actions towards unexplored states. A probability vector is a vector containing probabilities for each of its actions. In the probability vector $x = (x_1, x_2, ..., x_n)$, the probability x_i to choose the action i is given by Equation 2.8, where n is the number of actions in the set.

$$x_i = \begin{cases} (1-\epsilon) + (\epsilon/n), & \text{if } Q \text{ of } i \text{ is the highest} \\ \epsilon/n, & \text{otherwise} \end{cases} \tag{2.7}$$

2.6 Meta-Level Reasoning

Traditionally, *reasoning* is modeled as a decision cycle, in which the agent perceives environmental stimuli and responds to it with an appropriate action. The result of the actions performed in the environment (*ground-level*) is perceived by the agent (*object-level*), which responds with a new action, and so the cycle continues. This reasoning cycle is illustrated in Figure 2.3 [CR08].

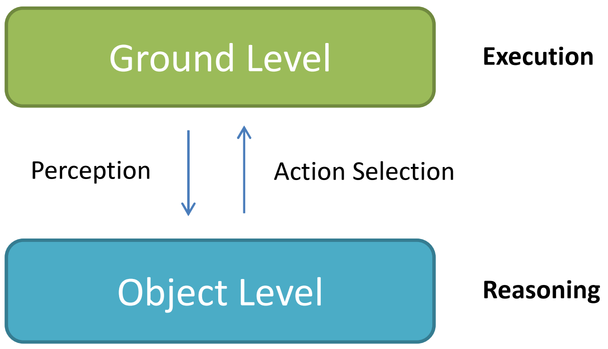


Figure 2.3 – Common cycle of perception and actions choice.

Meta-reasoning or *meta-level reasoning* is the process of explicitly reasoning about this reasoning cycle. It consists of both the control, and monitoring of the object-level reasoning, allowing an agent to adapt the reasoning cycle over time, as illustrated in Figure 2.4. This new cycle represents a high level reflection about its own reasoning cycle.

When meta-level reasoning is applied to learning algorithms, this gives rise to a new term: *meta-learning* [SD03, Doy02]. Meta-learning represents the concept of learning to learn, and the meta-learning level is generally responsible for controlling the parameters of the learning level. While learning at the object-level is responsible for accumulating experience about some task (e.g, take decisions in a game, medical diagnosis, fraud detection, etc.), learning at the meta-level is responsible for accumulating experience about learning algorithm itself. If learning at object-level is not suc-

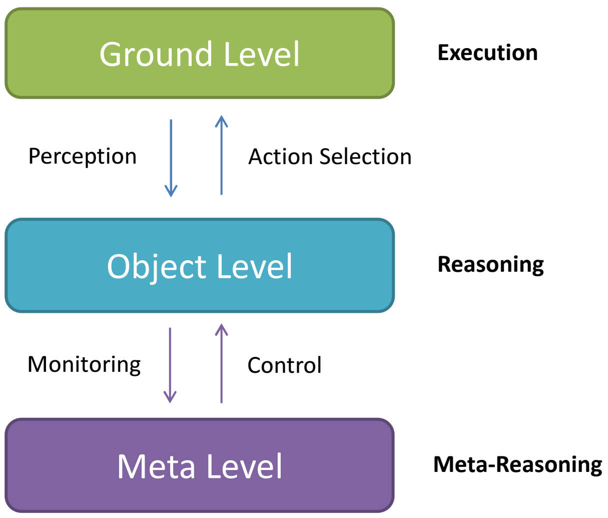


Figure 2.4 – Adding meta-level reasoning to the common cycle of perception and choice of actions.

ceeding in improving or maintaining performance, the meta-level learner takes the responsibility to adapt the object-level, in order to make it succeed. In other words, meta-learning helps solve important problems in the application of machine learning algorithms [VGCBS04], especially in dynamic environments.

2.7 StarCraft

Real-time strategy (RTS) are computer games in which multiple players control teams of characters and resources over complex simulated worlds where their actions occur simultaneously (thus there is no turn-taking between players). Players often compete over limited resources in order to strengthen their team and win the match. As such, RTS games are an interesting field for AI, because the state space is huge, actions are concurrent, and part of the game state is hidden from each player. Game-playing involves both the ability to manage each unit individually (*micro-management*), and a high-level strategy for building construction and resource gathering (*macro-management*).

StarCraft is an RTS created by *Blizzard Entertainment, Inc.*[1]. In this game, a player chooses between three different races to play (illustrated in Figure 2.5), each of which having different units, buildings and capabilities, and uses these resources to battle other players.

[1]StarCraft website in Blizzard Entertainment, Inc. http://us.blizzard.com/pt-br/games/sc/

Figure 2.5 – The different races in StarCraft.

The game consists on managing resources and building an army of different units to compete against the armies built by opposing players. Units in the game are created from structures, and there are prerequisites for building other units and structures. Consequently, one key aspect of the game is the order in which buildings and units are built, and good players have strategies to build them so that specific units are available at specific times for attack and defense moves. Such building strategies are called *build orders* or *BOs*. Strong BOs can put a player in a good position for the rest of the match. BOs usually need to be improvised from the first contact with the enemy units, since the actions become more dependent on knowledge obtained about the units and buildings available to the opponent [Hag12, CB11].

3. META-LEVEL REINFORCEMENT LEARNING

Having presented the state of the art on reinforcement learning and its related concepts, the following sections present our proposed solution.

3.1 Parameter Control

As we have seen in Section 2.4, the parameters used in the update rule of reinforcement learning influence the way state values are computed, and ultimately how a policy is generated. Therefore, the choice of parameters in reinforcement learning — such as α — can be crucial to the success in learning [SD03]. Consequently, there are different strategies to control and adjust these parameters.

When an agent does not know much about the environment, it needs to explore the environment with a high learning-rate to be able to quickly estimate the actual values of each state. However, a high learning-rate can either prevent the algorithm from converging, or lead to inaccuracies in the computed value of each state (e.g., a local maximum). For this reason, after the agent learns something about the environment, it should begin to modulate its learning-rate to ensure that either the state values converges, or that the agent overcomes local maxima. Consequently, maintaining a high learning-rate hampers the convergence of the Q-value, and Q-learning implementations often use a decreasing function for α as the policy is being refined. A typical way [SD03] to vary the α-value, is to start interactions with a value close to 1, and then decrease it over time toward 0. However, this approach is not effective for dynamic environments, since a drastic change in the environment when the learning-rate is close to 0 prevents the agent from learning the optimal policy in the changed environment.

3.2 Meta-Level Reasoning in Reinforcement Learning

The objective of meta-level reasoning is to improve the quality of decision making by explicitly reasoning about the parameters of the decision-making process and deciding how to change these parameters in response to the agent's performance. Consequently, an agent needs to obtain information about its own reasoning process to reason effectively at the meta-level. In this work, we consider the following processes used by our learning agent at each level of reasoning, and illustrate these levels in Figure 3.1:

- *ground-level* refers to the implementation of actions according to the MDP's policy;
- *object-level* refers to learning the parameters of the MDP and the policy itself;
- *meta-level* refers to manipulating the learning parameters used at the *object-level*;

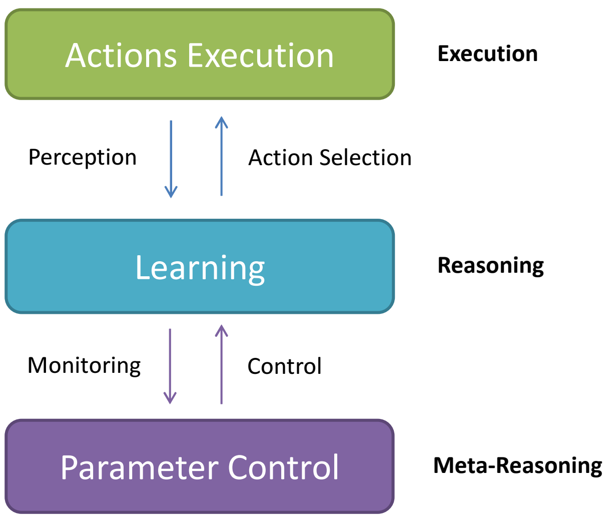


Figure 3.1 – Modeling the meta-level reasoning in reinforcement learning.

We have developed a number of strategies for meta-level reinforcement learning and evaluated them.

3.2.1 Meta-Level Reasoning in Q-Learning

Our approach to meta-level reasoning consists of varying the learning-rate (known as $\alpha-$value) to allow an agent to handle dynamic environments. More concretely, at the meta-level, we apply RL to learn the $\alpha-$value used as the learning-rate for object-level RL. In other words, we apply reinforcement learning to control the parameters of reinforcement learning.

The difference between RL applied at the meta-level and RL applied at the object-level is that, at the object-level, we learn Q-value for the state-action pair, increasing it when we have positive feedback and decreasing it when we have negative feedback. Conversely, at the meta-level, what we learn is the α-value, by decreasing it when we have positive feedback and increasing it when we have negative feedback — that is, making mistakes means we need to learn at a faster rate. Our approach to *meta-level reinforcement learning* is shown in Algorithm 3.1.

Meta-level reinforcement learning algorithm requires the same parameters as Q-learning: a state s, an action a and a reward R. In Line 1 we apply the RL update rule for the α-value used for the object-level Q-learning algorithm. At this point, we are learning the learning-rate, and as we saw, α decreases with positive rewards. We use a small constant step of 0.05 for the meta-level update rule and bound it between 0 and 1 (Lines 2–7) to ensure it remains a consistent learning-rate value for Q-learning. Such a small learning-rate at the meta-level aims to ensure that while we are constantly updating the object-level learning-rate, we avoid high variations. Finally, in Line 8 we use the standard update rule for Q-learning, using the adapted learning-rate. As our algorithm is nothing but a short sequence of mathematical operations, it is really efficient when it comes to time.

Algorithm 3.1 – Meta-Level Reasoning in Q-Learning

```
Require: $s, a, R$
1: $\alpha \leftarrow \alpha - (0.05 * R)$

2: if $\alpha < 0$ then
3:     $\alpha \leftarrow 0$
4: end if
5: if $\alpha > 1$ then
6:     $\alpha \leftarrow 1$
7: end if

8: $Q(s,a) \leftarrow Q(s,a) + (\alpha * R)$
```

Thus, it is able to execute in few clock cycles and could be utilized in real-time after each action execution.

3.2.2 Meta-Level Reasoning in Exploration Policy

Since we are modifying the learning-rate based on the feedback obtained by the agent, and increasing it when the agent detects that its knowledge is no longer up to date, we can also use this value to guide the exploration policy. Thus, we also modify the $\epsilon-$greedy action selection algorithm. Instead of keeping the exploration-rate ($\epsilon-$value) constant, we apply the same meta-level reasoning to the $\epsilon-$value, increasing the exploration-rate, whenever we find that the agent must increase its learning-rate — the more the agent wants to learn, the more it wants to explore; if there is nothing to learn, there is nothing to explore. To accomplish this, we had first defined the exploration-rate as been always equal to the learning-rate:

$$\epsilon \leftarrow \alpha$$

This solution has proved not to work well, since there are cases when the learning-rate and exploration-rate are maximized in 1, causing the agent to choose all its actions randomly and preventing the convergence of learning. When the exploration-rate is near 1, the agent chooses more random actions than learned actions, which normally makes the agent commit more mistakes, preventing the learning-rate from converging to 0. To solve this, we defined the exploration-rate to be half of learning-rate. This makes the highest exploration-rate be 0.5, when the learning-rate is maximized in 1, while still maintaining a exploration-rate of 0 when learning-rate is 0.

$$\epsilon \leftarrow \alpha/2$$

Our approach to *meta-level exploration policy* is shown in Algorithm 3.2.

Algorithm 3.2 – Meta-Level Exploration Policy

```
Require: $V_a$ (vector with possible actions), $\alpha$ (learning-rate)
1: $a_{max} \leftarrow \max_{a \in V_a}$
2: $r \leftarrow rand(0,1)$
3: $\epsilon \leftarrow \alpha/2$
4: if $r > \epsilon$ then
5:     return $a_{max}$
6: else
7:     return $any(V_a)$
8: end if
```

The Meta-level exploration policy algorithm requires two parameters: learning-rate (α) and a vector with all the possible actions (V_a). First, it selects the max-action — action with highest Q-value for current state — in Line 1. Next it gets a random value between 0 and 1 (in Line 2) using it as random factor. Then, in Line 3 it calculates the exploration-rate (ϵ). Finally, between lines 4 and 8, if the random factor was higher than exploration-rate, it returns the max-action; else, it just returns any possible action.

3.2.3 Meta-Level Reinforcement Learning

By using meta-level reasoning, we have improved the classical reinforcement learning. We use meta-level reasoning in the two approaches — Q-learning and exploration policy — together to create *meta-level reinforcement learning*. With both, we cover the two parts of our problem: *learning* and *exploration*.

4. EXPERIMENTS AND RESULTS

In this section, we detail our implementation of meta-level reinforcement learning and its integration to the *Starcraft* game, followed by our experiments and their results. These results concern the complete implementation of this work. For results of our first partial implementation, please see [MM13].

4.1 Interacting with StarCraft

The first challenge in implementing the algorithm is the integration of our learning algorithm to the proprietary code from Starcraft, since we cannot directly modify its code and need external tools to do this. In the case of StarCraft, community members developed the *BWAPI*, which allows us to inject code into the existing game binaries. The *BWAPI (Brood War Application Programming Interface)*[1] enables the creation and injection of artificial intelligence code into StarCraft. BWAPI was initially developed in *C++*, and later ported to other languages like *Java*, *C#* and *Python*, and divides StarCraft in 4 basic types of object:

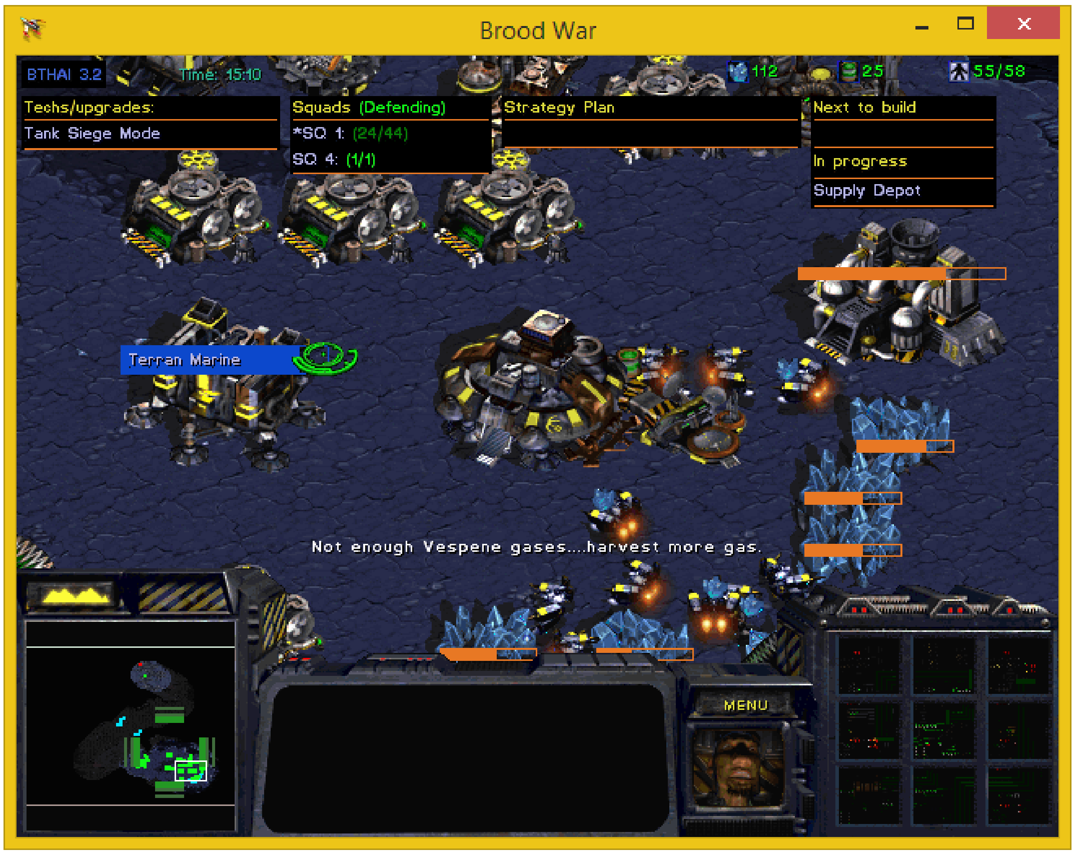
- *Game:* manages information about the current game being played, including the position of known units, location of resources, etc.;
- *Player:* manages the information available to a player, such as: available resources, buildings and controllable units;
- *Unit:* represents a piece in the game, either mineral, construction or combat unit;
- *Bullet:* represents a projectile fired from a ranged unit;

Since the emergence of BWAPI in 2009, StarCraft has drawn the attention of researchers and an active community of bot programming has emerged [BC12]. For our implementation, we modified the open source bot *BTHAI* [Hag12], adding a high-level strategy learning component to it[2]. Figure 4.1 shows a screenshot of a game where one of the players is controlled by BTHAI, notice the additional information overlaid on the game interface.

4.2 A Reinforcement Learning Approach for StarCraft

Following the approach used by [AS10], our approach focuses on learning the best high-level strategy to use against an opponent. We assume here that the agent will only play as Terran, and will be able to choose any one of the following strategies (units can be see at Figure 4.2):

[1]An API to interact with StarCraft: BroodWar http://code.google.com/p/bwapi/
[2]The source code can be fount at: https://github.com/jieverson/BTHAIMOD



Not enough Vespene gases....harvest more gas.

Figure 4.1 – BTHAI bot playing StarCraft: Brood War.

- *Marine Rush:* is a very simple Terran strategy that relies on quickly creating a specific number of workers (just enough to maintain the army) and then spending all the acquired resources on the creation of Marines (the cheapest Terran battle unit) and making an early attack with a large amount of units.

- *Wraith Harass:* is similar, but slightly improved, Marine rush that consists of adding a mixture of 2–5 Wraiths (a relatively expensive flying unit) to the group of Marines. The Wraith's mission is to attack the opponent from a safe distance, and when any of the Wraiths are in danger, use some Marines to protect it. Unlike the Marine Rush, this strategy requires strong micromanagement, making it more difficult to perform.

- *Terran Defensive:* consists of playing defensively and waiting for the opponent to attack before counterattacking. Combat units used in this strategy are Marines and Medics (a support unit that can heal biological units), usually defended by a rearguard of Siege Tanks.

- *Terran Defensive FB:* is slightly modified version of the Terran Defensive strategy, which replaces up to half of the Marines by Firebats — a unit equipped with flamethrowers that is especially strong against non-organic units such as aircrafts, tanks and most of Protoss' units.

- *Terran Push:* consists of creating approximately five Siege Tanks and a large group of Marines, and moving these units together through the map in stages, stopping at regular intervals to regroup. Given the long range of the Tank's weapons, opponents will often not perceive their approach until their units are under fire, however, this setup is vulnerable to air counterattack units.

Figure 4.2 – Most common Terran Units: SCV, Marine, Wraith, Medic and Siege Tank

After each game, the agent observes the end result (victory or defeat), and uses this feedback to learn the best strategy. If the game is played again, the learning continues, so we can choose the strategy with the highest value for the current situation. If the agent perceives, at any time, that the strategy ceases to be effective — because of a change in the opponent's strategy, map type, race or other factors — the agent is able to quickly readapt to the new conditions, choosing a new strategy.

4.3 Experiments with StarCraft

To demonstrate the applicability of our approach we have designed an experiment whereby a number of games are played against a single opponent that can play using different AI bot strategies. We seek to determine if our learning methods can adapt an agent's policy when the AI bot changes. Each game was played in a small two-player map (*Fading Realm*) using the maximum game speed (since all players were automated). The game was configured to start another match as soon as the current one ends. For the experiment, all the Q-values are initialized to 0, and the learning-rate (α) is initialized to 0.5. Our experiment consisted of playing a total of 100 matches where one of the players is controlled by an implementation of our meta-learning agent. In the first 30 matches, the opponent have played a fixed Terrain policy provided by the game and in subsequent matches, we have changed the opponent policy to the fixed Protoss policy provided by the game. It is worth noting that our method used very little computation time–it runs in real time, using accelerated game speed (for matches between two bots).

4.3.1 Applying Classical Q-Learning

First of all, we have applied the classical Q-learning, with the purpose of comparing with Meta-Level Reinforcement Learning methods.

Figure 4.3 illustrates the variation of the strategies Q-values over each game execution using the *classical Q-Learning* method. We can see that the *Marine Rush* strategy was optimal

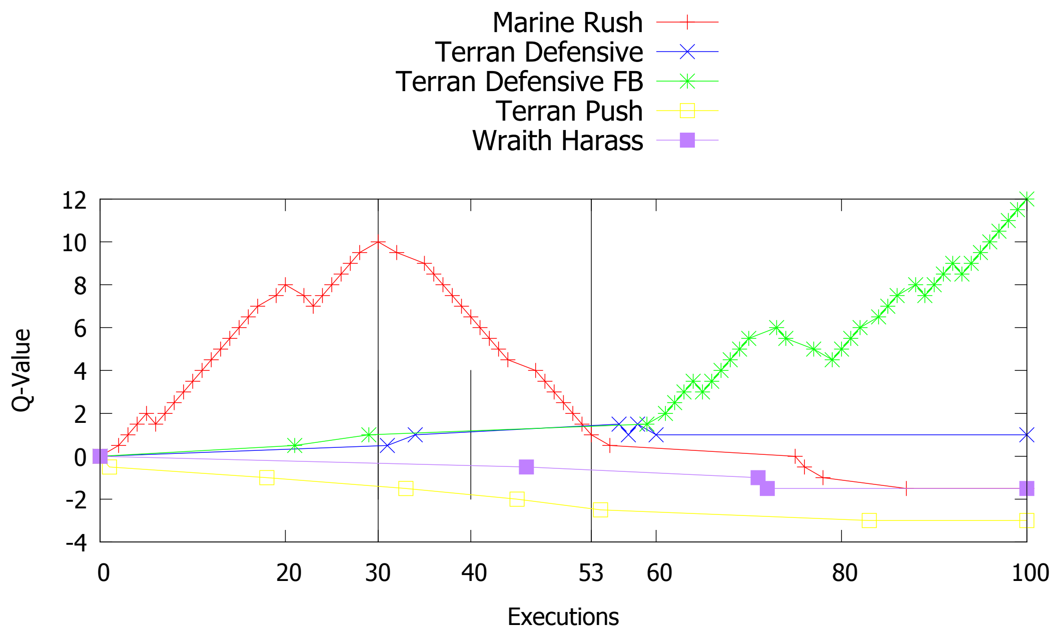


Figure 4.3 – Strategies Q-value over time using the classical Q-Learning method.

against the first opponent policy, while the *Terrain Push* has proven to be the worst. When the opponent changes its policy in the execution 30, we can see the Q-value of *Marine Rush* decreases, resulting in an increase in exploration. After the execution 53, we notice that the *Terrain Defensive FB* strategy stood out from the others, although the basic *Terrain Defensive* strategy has shown to yield good results too. *Wraith Harass* and *Marine Rush* seem to lose to the second opponent policy, and *Terrain Push* remains the worst strategy.

4.3.2 Applying Meta-Level Reinforcement Learning

Next, Figure 4.4 illustrates the variation of the strategies' Q-values over each game execution using the proposed *Meta-Level Reinforcement Learning*. The results look similar to the last experiment, with some subtle differences:

- Using the Meta-Level Learning, Q-values stabilize and stop growing at some point. Figure 4.4 shows that the values stabilize between executions 10 and 30, as well as between the executions 60 and 100. It occurs because the Q-values stop getting higher when the learning-rate becomes 0;

- The classical Q-Learning method has reached higher Q-values (Figure 4.3) such as 10 on the execution 30, and 12 at execution 100, while the Meta-Level strategy (Figure 4.4) has reached just 2 on the execution 30 and 4 at execution 100;

- Because of the lower Q-values, the Meta-Level method proved to respond faster to opponent changes. This can be see after the execution 30. Using classical Q-Learning, the agent only discovered the new best strategy at the execution 53, while the Meta-Level Reinforcement Learning discovered it at execution 40: it learned twice as fast. If the opponent change was made after the execution 30, this difference would be larger.

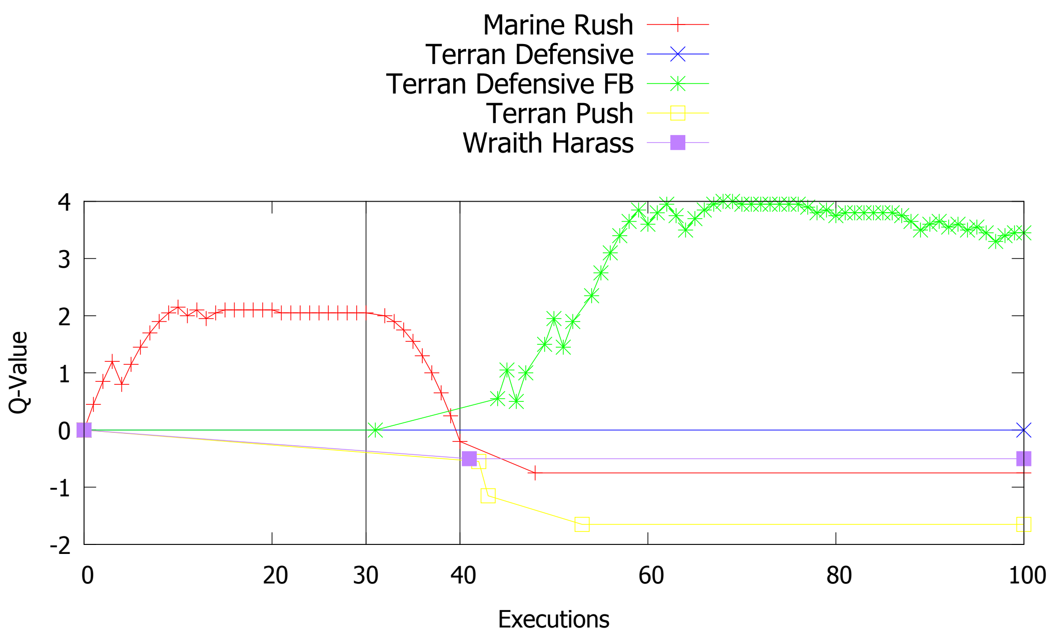


Figure 4.4 – Strategies Q-value over time using Meta-Level Reinforcement Learning.

In the Meta-Level Reinforcement Learning, as the agent learns, its learning-rate tends to decrease towards 0, which means that the agent has nothing to learn. After the change in opponent policy (at game execution 30), we expected the learning-rate to increase, denoting that the agent is starting to learn again, which was indeed the case, as illustrated by the graph of Figure 4.5.

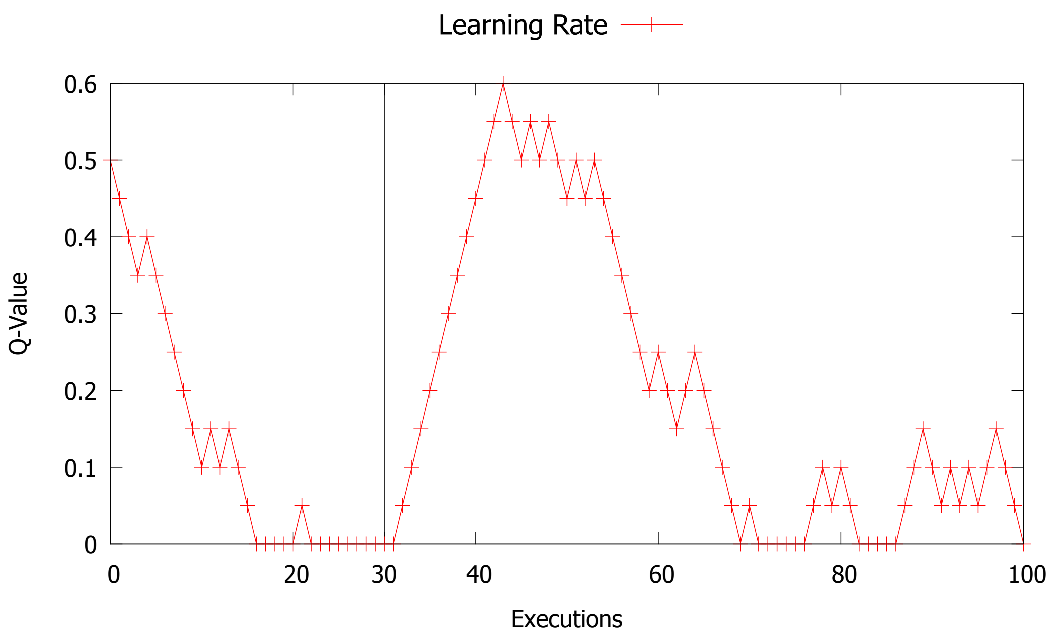


Figure 4.5 – Learning-rate variation over time using Meta-Level Reinforcement Learning.

The learning-rate should remain greater than 0 until the RL algorithm converges to the optimal policy, and then start decreasing towards 0. We note that, although the learning-rate may vary between 0 and 1, it has never gone beyond 0.6 in the executions we performed.

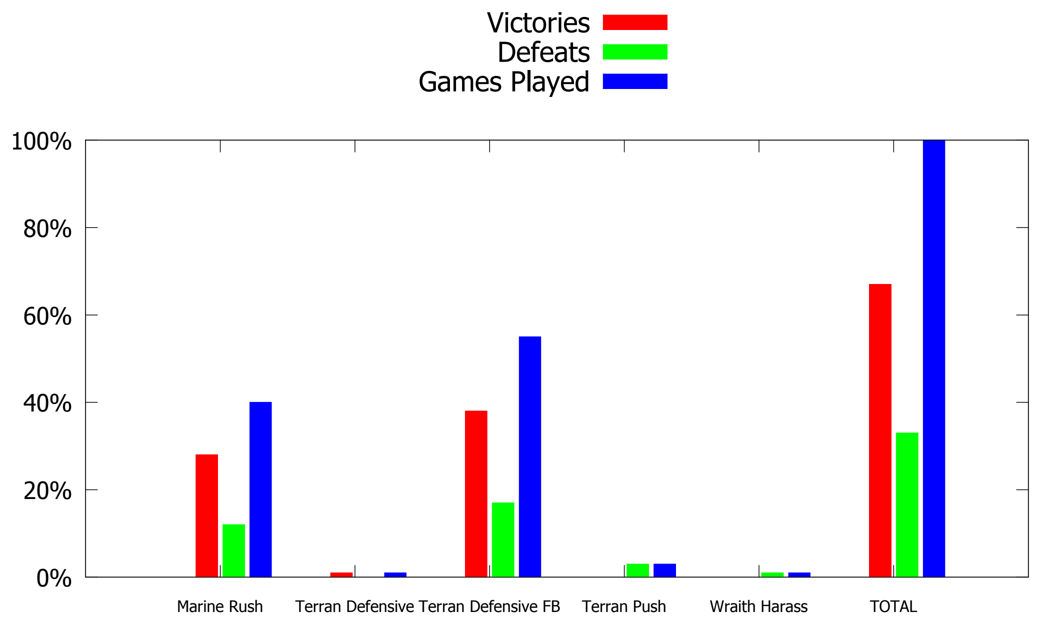


Figure 4.6 – Comparison between the number of victories and defeats of each strategy.

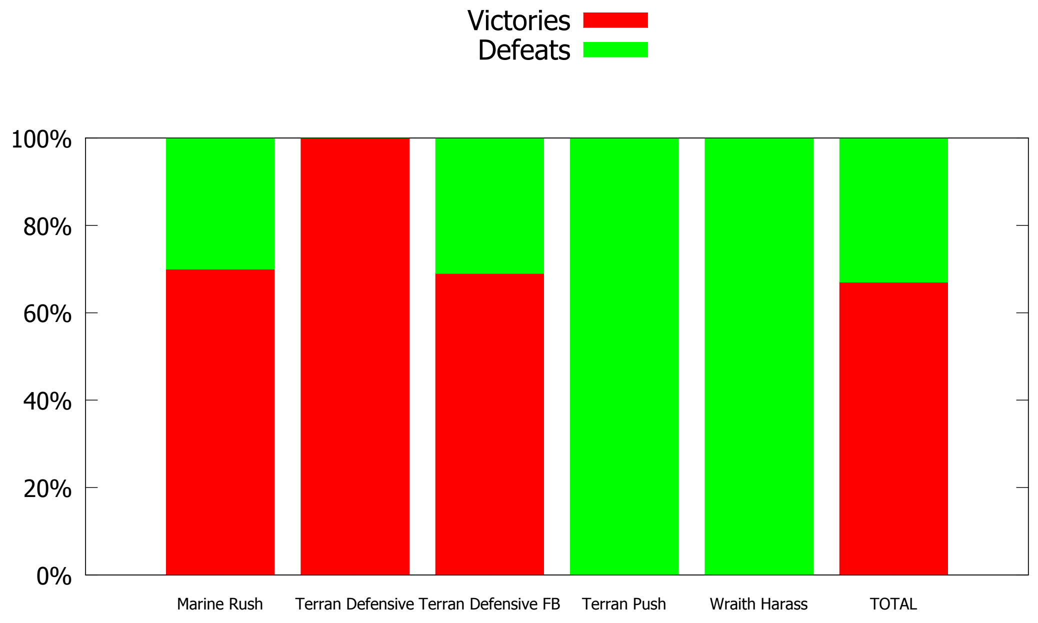


Figure 4.7 – Graphic that presents a comparation between the win rate of each strategy.

The results obtained are also illustrated in the graph of Figure 4.6 and Figure 4.7, which shows that our meta-learning agent consistently outperforms fixed opponents. Moreover, we can see that the agent quickly learns the best strategy to win against a fixed policy opponent when its strategy changes.

5. CONCLUSION

In this work we have developed a reinforcement learning mechanism for high-level strategies in RTS games that is able to cope with the opponent abruptly changing its play style. To accomplish this, we have applied meta-level reasoning techniques over the already known RL strategies, so that we learn how to vary the parameters of reinforcement learning allowing the algorithm to "de-converge" when necessary. The aim of our technique is to learn when the agent needs to learn faster or slower. Although we have obtained promising initial results, our approach was applied just for high-level strategies, and the results were collected using only the strategies built into the BTHAI library for Starcraft control. To our knowledge, ours is the first approach to mix meta-level reasoning and reinforcement learning that applies RL to control the parameters of RL. Furthermore, we have modified the way exploration policy is done, introducing a new concept to vary exploration policy using learning-rate. We had part of this work published in [MM13].

Results have shown that this meta-level strategy can be a good solution to find high-level strategies. The meta-learning algorithm we developed is not restricted to StarCraft and can be used in any game in which the choice of different strategies may result in different outcomes (victory or defeat), based on the play style of the opponent.

In the future, we aim to apply this approach to low-level strategies, such as learning detailed *build orders* or to micro-manage battles. In this work we have not experimented the use of model based learning, which we think is another good option for future works. Other approaches to try would be to vary just exploration-rate (ϵ), keeping a static learning-rate (α). Given our initial results, we believe that meta-level reinforcement learning is a useful technique in game AI control that can be used on other games, at least at strategic level.

BIBLIOGRAPHY

[AS10] Amato, C.; Shani, G. "High-level reinforcement learning in strategy games". In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems, 2010, pp. 75–82.

[BC12] Buro, M.; Churchill, D. "Real-time strategy game competitions", *AI Magazine*, vol. 33–3, 2012, pp. 106–108.

[Bel57] Bellman, R. "Dynamic Programming". Princeton University Press, Princeton, New Jersey, 1957.

[CB11] Churchill, D.; Buro, M. "Build order optimization in starcraft". In: Proceedings of the Seventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 2011, pp. 14–19.

[CR08] Cox, M. T.; Raja, A. "Metareasoning: A manifesto". In: Proceedings of AAAI 2008 Workshop on Metareasoning: Thinking about Thinking, 2008, pp. 106–112.

[dOBdS$^+$06] de Oliveira, D.; Bazzan, A. L.; da Silva, B. C.; Basso, E. W.; Nunes, L.; Rossetti, R.; de Oliveira, E.; da Silva, R.; Lamb, L. "Reinforcement learning based control of traffic lights in non-stationary environments: A case study in a microscopic simulator." In: EUMAS, 2006.

[Doy02] Doya, K. "Metalearning and neuromodulation", *Neural Networks*, vol. 15–4, 2002, pp. 495–506.

[Dre02] Dreyfus, S. "Richard bellman on the birth of dynamic programming", *Operations Research*, 2002, pp. 48–51.

[GHG04] Graepel, T.; Herbrich, R.; Gold, J. "Learning to fight". In: Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education, 2004, pp. 193–200.

[Gho04] Ghory, I. "Reinforcement learning in board games", Technical Report CSTR-04-004, University of Bristol, 2004.

[GRO03] Guelpeli, M.; Ribeiro, C.; Omar, N. "Utilização de aprendizagem por reforço para modelagem autônoma do aprendiz em um tutor inteligente". In: Anais do Simpósio Brasileiro de Informática na Educação, 2003, pp. 465–474.

[Hag12] Hagelbäck, J. "Potential-field based navigation in starcraft". In: Proceedings of the 2012 IEEE Conference on Computational Intelligence and Games (CIG), 2012, pp. 388–393.

[KLM96] Kaelbling, L.; Littman, M.; Moore, A. “Reinforcement learning: A survey”, *Arxiv preprint cs/9605103*, vol. 4, 1996, pp. 237–285.

[MA93] Moore, A. W.; Atkeson, C. G. “Prioritized sweeping: Reinforcement learning with less data and less time”, *Machine Learning*, vol. 13–1, 1993, pp. 103–130.

[Mit64] Mitten, L. “Composition principles for synthesis of optimal multistage processes”, *Operations Research*, vol. 12–4, 1964, pp. 610–619.

[ML10] Mohan, S.; Laird, J. E. “Relational reinforcement learning in infinite mario.” In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, 2010, pp. 1953–1954.

[MM13] Maissiat, J.; Meneguzzi, F. “Adaptive high-level strategy learning in starcraft”. In: Proceedings of the SBGames conference on Computing, 2013.

[Mor82] Morin, T. “Monotonicity and the principle of optimality”, *Journal of Mathematical Analysis and Applications*, vol. 88–2, 1982, pp. 665–674.

[Put94] Puterman, M. “Markov decision processes: Discrete stochastic dynamic programming”. John Wiley & Sons, Inc., 1994.

[RGK09] Rodrigues Gomes, E.; Kowalczyk, R. “Dynamic analysis of multiagent q-learning with ε-greedy exploration”. In: Proceedings of the 26th Annual International Conference on Machine Learning, 2009, pp. 369–376.

[RN09] Russell, S.; Norvig, P. “Artificial Intelligence: A Modern Approach”. Prentice Hall, 2009, vol. 2.

[SB98] Sutton, R.; Barto, A. “Reinforcement learning: An introduction”. Cambridge Univ Press, 1998, vol. 1.

[SC96] Sandholm, T.; Crites, R. “Multiagent reinforcement learning in the iterated prisoner's dilemma”, *Biosystems*, vol. 37–1-2, 1996, pp. 147–166.

[SD03] Schweighofer, N.; Doya, K. “Meta-learning in reinforcement learning”, *Neural Networks*, vol. 16–1, 2003, pp. 5–9.

[SSK05] Stone, P.; Sutton, R.; Kuhlmann, G. “Reinforcement learning for robocup soccer keepaway”, *Adaptive Behavior*, vol. 13–3, 2005, pp. 165–188.

[Sut91] Sutton, R. S. “Dyna, an integrated architecture for learning, planning, and reacting”, *ACM SIGART Bulletin*, vol. 2–4, 1991, pp. 160–163.

[Tay11] Taylor, M. “Teaching reinforcement learning with mario: An argument and case study”. In: Proceedings of the Second Symposium on Educational Advances in Artifical Intelligence, 2011, pp. 1737–1742.

[TK02] Tesauro, G.; Kephart, J. O. “Pricing in agent economies using multi-agent q-learning.”, *Autonomous Agents and Multi-Agent Systems*, vol. 5–3, 2002, pp. 289–304.

[UJG08] Ulam, P.; Jones, J.; Goel, A. K. “Combining model-based meta-reasoning and reinforcement learning for adapting game-playing agents.” In: Proceedings of the Fourth AAAI Conference on AI in Interactive Digital Environment, 2008.

[VGCBS04] Vilalta, R.; Giraud-Carrier, C. G.; Brazdil, P.; Soares, C. “Using meta-learning to support data mining”, *International Journal of Computer Science & Applications*, vol. 1–1, 2004, pp. 31–45.

[WD92] Watkins, C. J.; Dayan, P. “Technical note: Q-learning”. In: *Reinforcement Learning*, Springer, 1992, pp. 55–68.