# Adaptive High-Level Strategy Learning in StarCraft

Jiéverson Maissiat
Faculdade de Informática
Pontifícia Universidade Católica
do Rio Grande do Sul (PUCRS)
Email: contact@jieverson.com

Felipe Meneguzzi
Faculdade de Informática
Pontifícia Universidade Católica
do Rio Grande do Sul (PUCRS)
Email: felipe.meneguzzi@pucrs.br

***Abstract*—Reinforcement learning (RL) is a technique to compute an optimal policy in stochastic settings whereby, 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 when human opponents are possible. Consequently, in this paper, 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.**

## I. Introduction

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 [9]. A number of algorithms and strategies for reinforcement learning have been proposed in the literature [15], [7], 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 [17], [10]. However, traditional reinforcement learning often assumes that the environment remains static throughout the learning process so that when the learning algorithm converges. Under the assumption that the environment remains static over time, when the 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 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 paper, we apply the concept of meta-level reasoning [4], [19] to reinforcement learning [14] 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.

This paper is organized as follows: in Section II we review the main concepts used in required for this paper: the different kinds of environments (II-A), some concepts of machine learning (II-B) and reinforcement learning (II-C); in Section III we explain the StarCraft game domain, and in Section IV we describe our solution. Finally, we demonstrate the effectiveness of our algorithms through empirical experiments and results in Section V.

## II. Background

### A. 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 [12]: 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.

## B. *Machine Learning*

An agent is said to be learning if it improves its performance after observing the world around it [12]. 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 [12] 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.

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 paper, we focus in reinforcement learning technique.

## C. *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 [16]. Figure 1 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 ensure 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 [12]. 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 on every iteration of the learning algorithm. A passive agent has 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

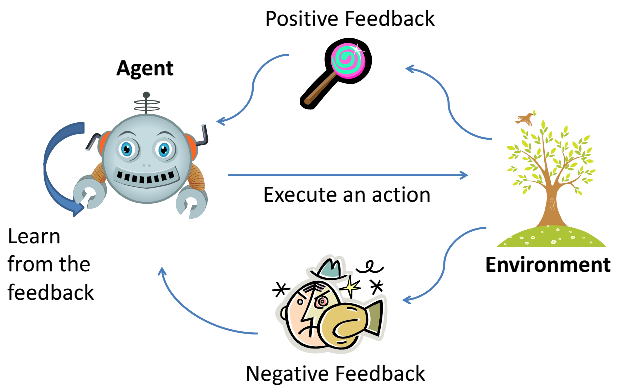

Fig. 1. Model to describe the process of reinforcement learning.

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 [12], 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 values [6].

*1) Q-Learning:* Depending on the assumptions about the agent knowledge prior to learning, different algorithms are 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 following adjustment function [1].

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

Where,

- $s$ represents the current state of the world;
- $a$ represents the 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;
- $\alpha$ 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;

- $\gamma$ is the *discount factor*, which determines the importance of future rewards — a factor of 0 makes the agent opportunistic [14] by considering only the current reward, while a factor of 1 makes the agent consider future rewards, seeking to increase their long-term rewards;

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 following rule:

$$\pi^{\approx}(s) = \arg\max_a Q(s,a)$$

In dynamic environments, Q-learning does not guarantee convergence to the optimal policy. 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 convergence [13], [18], [1]. 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) 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 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 optimal overall (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 [1].

In this paper we use an exploration mechanism known as $\epsilon$-greedy [11]. This mechanism has a probability $\epsilon$ to select a random action, and a probability $1-\epsilon$ to select the optimal action known so far — which 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. In the probability vector $x = (x_1, x_2, ..., x_n)$, the probability $x_i$ to choose the action $i$ is given by:

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

where $n$ is the number of actions in the set.

### D. Meta-Level Reasoning

Traditionally, *reasoning* is modeled as a decision cycle, in which the agent perceives environmental stimulus 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 [4].

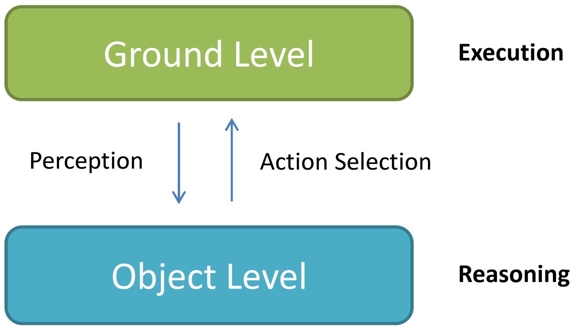

Fig. 2. Common cycle of perception and actions choice.

*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 3. This new cycle represents a high level reflection about its own reasoning cycle.

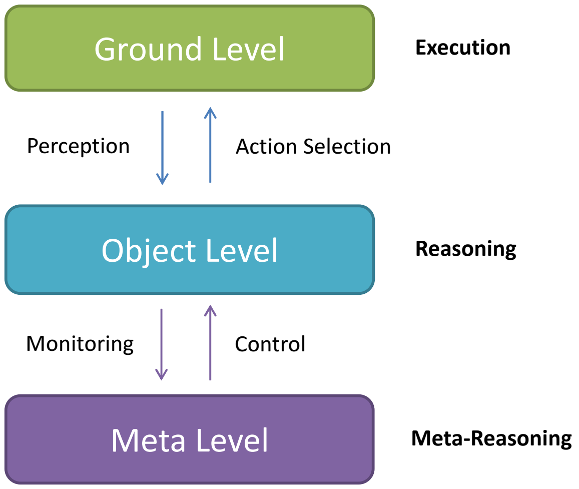

Fig. 3. Adding meta-level reasoning to the common cycle of perception and choice of actions.

When meta-level reasoning is applied to learning algorithms, this gives rise to a new term: *meta-learning* [14], [5]. 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 succeeding 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 [20], especially in dynamic environments.

## III. StarCraft

*Real-time strategy* (RTS) games are computer games in which multiple players control teams of characters and resources over complex simulated worlds where their actions occur simultaneously (so 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 the AI, because the state space is huge, actions are concurrent, and part of the game state is hidden from each player. Game-play 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 4), each of which having different units, buildings and capabilities, and uses these resources to battle other players, as shown in Figure 5. 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. These 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 [8], [3].

Fig. 4. StarCraft: Brood War – Race selection screen.

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

Fig. 5. StarCraft: Brood War – Batttle Scene.

## IV. Meta-Level Reinforcement Learning

### A. Parameter Control

As we have seen in Section II-C, the parameters used in the update rule of reinforcement learning influence how the state values are computed, and ultimately how a policy is generated. Therefore, the choice of the parameters in reinforcement learning — such as $\alpha$ and $\gamma$ — can be crucial to success in learning [14]. 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 learn 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 converge, 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 $\alpha$ as the policy is being refined. A typical way [14] to vary the $\alpha$-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 with a learning-rate close to 0 prevents the agent from learning the optimal policy in the changed environment.

### B. Meta-Level Reasoning on 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 paper, we consider the following processes used by our learning agent at each level of reasoning, and illustrate these levels in Figure 6:

- *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 in *object-level*;

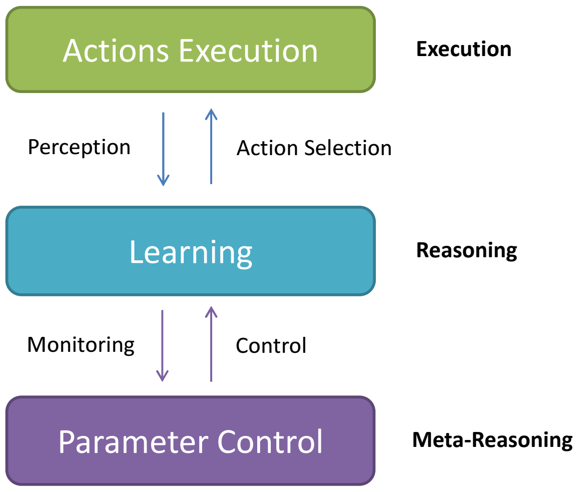

Fig. 6. Modeling the meta-level reasoning in reinforcement learning.

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 at the 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 action-state pair, increasing it when we have positive feedback and decreasing it when we have negative feedback. Conversely, at the meta-level, what we learn in the $\alpha$-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 1.

**Algorithm 1** Meta-Level Reinforcement 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: $Q(s,a) \leftarrow Q(s,a) + (\alpha * R)$
```

The 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 $\alpha$-value used for the object-level Q-learning algorithm. At this point, we are learning the learning-rate, and as we saw, $\alpha$ decreases with positive rewards. We use a small constant learning-rate 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 the algorithm is nothing but a sequence of mathematical operations, it is really efficient when it comes to time. Thus, it is able to execute in few clock cycles and could be utilized in real-time after each action execution.

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 exploitation-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 define the exploitation-rate as been always equal to the learning-rate:

$$\epsilon = \alpha$$

## V. 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.

### A. *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)*[2] 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

[2]An API to interact with StarCraft: BroodWar http://code.google.com/p/bwapi/

of bot programming has emerged [2]. For our implementation, we modified the open source bot *BTHAI* [8], adding a high-level strategy learning component to it[3]. Figure 7 shows a screenshot of a game where one of the players is controlled by BTHAI, notice the additional information overlaid on the game interface.

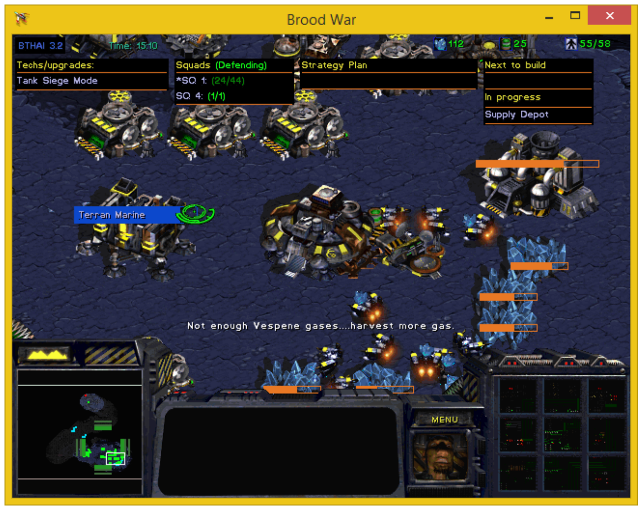

Fig. 7. BTHAI bot playing StarCraft: Brood War.

## B. A Reinforcement Learning Approach for StarCraft

Following the approach used by [1], 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:

- *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.

[3]The source code can be fount at: https://github.com/jieverson/BTHAIMOD

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.

## C. 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 its 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 ($\alpha$) is initialized to 0.5. Our experiment consisted of playing the game a total of 138 matches where one of the players is controlled by an implementation of our meta-learning agent. In the first 55 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).

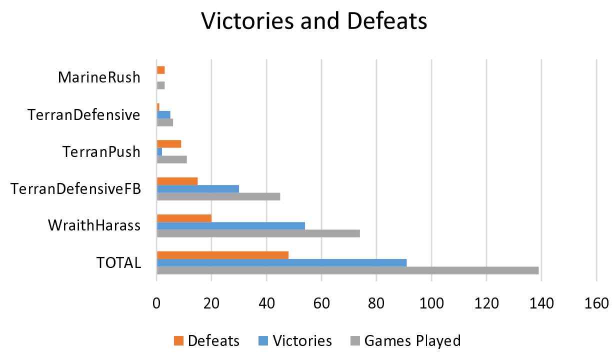

Fig. 8. Comparison between the number of victories and defeats of each strategy.

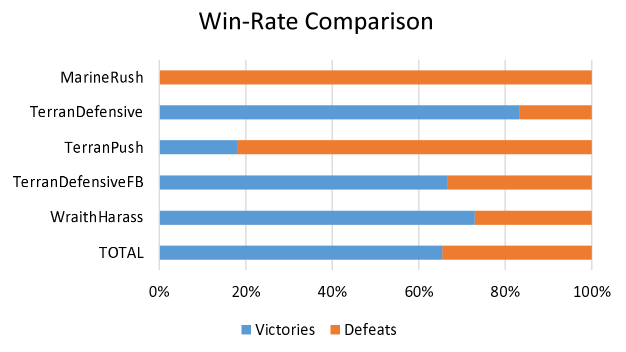

Fig. 9. Graphic that presents a comparation between the win rate of each strategy.

The results obtained are illustrated in the graph of Figure 8 and Figure 9, 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. As it learns, its learning-rate should tend to decrease towards 0, which means that the agent has nothing to learn. After the change in opponent policy (at game execution 55), 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 10. The learning rate should remain above 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 0.7 in the executions we performed.

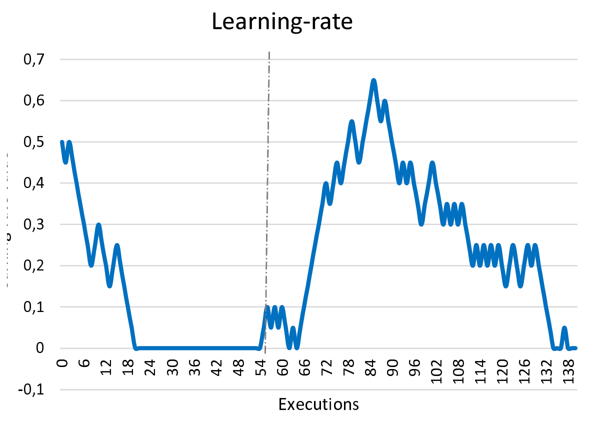

Fig. 10. Learning-rate variation over time.

Finally, the graphic in Figure 11 illustrates the variation of the strategies Q-values over each game execution. We can see that the *Wraith Harass* strategy was optimal against the first opponent policy, while the *Terrain Push* has proven to be the worst. When the opponent changes its policy, we can see the Q-value of *Wraith Harass* decreases, resulting in an increase in exploration. After the execution 85, 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* shows remain the worst strategy.

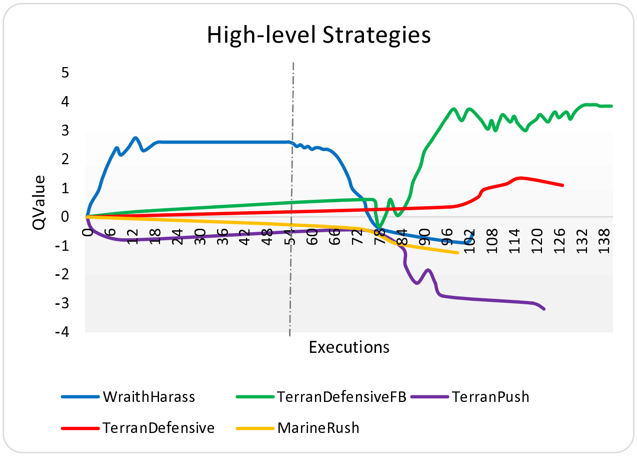

Fig. 11. Strategies Q-Value over time.

## VI. Conclusion

In this paper 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. The 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. 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 a strategic level.

## Acknowledgment

The authors would like to thank the members of the BTHAI and BWAPI groups for making available and documenting the tools that made this work possible.

## References

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

[2] M. Buro and D. Churchill. Real-time strategy game competitions. *AI Magazine*, 33(3):106–108, 2012.

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

[4] M. T. Cox and A. Raja. Metareasoning: A manifesto. In *Proceedings of AAAI 2008 Workshop on Metareasoning: Thinking about Thinking*, pages 106–112, 2008.

[5] K. Doya. Metalearning and neuromodulation. *Neural Networks*, 15(4):495–506, 2002.

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

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

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

[9] L. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey. *Arxiv preprint cs/9605103*, 4:237–285, 1996.

[10] S. Mohan and J. E. Laird. Relational reinforcement learning in infinite mario. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 1953–1954, 2010.

[11] E. Rodrigues Gomes and R. Kowalczyk. Dynamic analysis of multiagent q-learning with $\varepsilon$-greedy exploration. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 369–376. ACM, 2009.

[12] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, volume 2. Prentice Hall, 2009.

[13] T. Sandholm and R. Crites. Multiagent reinforcement learning in the iterated prisoner's dilemma. *Biosystems*, 37(1-2):147–166, 1996.

[14] N. Schweighofer and K. Doya. Meta-learning in reinforcement learning. *Neural Networks*, 16(1):5–9, 2003.

[15] P. Stone, R. Sutton, and G. Kuhlmann. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[16] R. Sutton and A. Barto. *Reinforcement learning: An introduction*, volume 1. Cambridge Univ Press, 1998.

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

[18] G. Tesauro and J. O. Kephart. Pricing in agent economies using multi-agent q-learning. *Autonomous Agents and Multi-Agent Systems*, 5(3):289–304, 2002.

[19] P. Ulam, J. Jones, and A. K. Goel. Combining model-based metareasoning and reinforcement learning for adapting game-playing agents. In *Proceedings of the Fourth AAAI Conference on AI in Interactive Digital Environment*, 2008.

[20] R. Vilalta, C. G. Giraud-Carrier, P. Brazdil, and C. Soares. Using metalearning to support data mining. *International Journal of Computer Science & Applications*, 1(1):31–45, 2004.