Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques
Universitat de Barcelona

Q-LEARNING IN RTS GAME'S MICRO-MANAGEMENT

Angel Camilo Palacios Garzón

Directors: Mike Preuss & Jesús Cerquides

Realitzat a: Departament de Matemàtica Aplicada i Anàlisi. UB

Abstract

The purpose of this Project is to implement the one-step Q-Learning algorithm and a similar version using linear function approximation in a combat scenario in the Real-Time Strategy game *Starcraft: Broodwar*™. First, there is a brief description of Real-Time Strategy games, and particularly about Starcraft, and some of the work done in the field of Reinforcement Learning. After the introduction and previous work are covered, a description of the Reinforcement Learning problem in Real-Time Strategy games is shown. Then, the development of the Reinforcement Learning agents using Q-Learning and Approximate Q-Learning is explained. It is divided into three phases: the first phase consists of defining the task that the agents must solve as a Markov Decision Process and implementing the Reinforcement Learning agents. The second phase is the training period: the agents have to learn how to destroy the rival units and avoid being destroyed in a set of training maps. This will be done through exploration because the agents have no prior knowledge of the outcome of the available actions. The third and last phase is testing the agents' knowledge acquired in the training period in a different set of maps, observing the results and finally comparing which agent has performed better. The expected behavior is that both Q-Learning agents will learn how to kite (attack and flee) in any combat scenario. Ultimately, this behavior could become the micro-management portion of a new Bot or could be added to an existing bot.

Keywords: Q-Learning, Reinforcement Learning, Real-Time Strategy, Starcraft, BWAPI, Micro-Management.

Index

1. Introduction

Since their first appearance in the 1980s Real-Time Strategy (hereafter, RTS) games have represented a big challenge for Artificial Intelligence (hereafter, AI) developers, and it remains an open problem today. There has been a pleather of research and improvements, especially after Buro's call for AI research in RTS games [1], but, unlike other games like chess or checkers, even the state of the art RTS game agents are far from playing at human level. Some of the problems faced by AI developers in RTS games are the large state space of the game, incomplete information, decision making under uncertainty and adversarial planning. These problems can also be found in real life situations such as air traffic controlling, automated vehicles, exploration with drones or weather prediction to mention just a few. Thus, RTS games are useful as finite discrete simplifications of the real world and the achievements in RTS games research are beneficial to other fields.

The goal of this project is to develop an AI agent (also known as a Bot) that can learn an efficient way of engaging the enemy in a small combat scenario using Reinforcement Learning (hereafter, RL) algorithm Q-Learning and its similar version with linear function approximation. In particular, the Bot should learn the 'Kiting' technique (also known as 'hit and run'), a micro-technique consisting of attacking the enemy and fleeing repeatedly. This technique should improve the survival rate of range units compared to the built-in AI survival rate.

Once the bot is implemented it will play several maps divided into two parts; the training period and the test period. All of the learning process occurs during the training period, and it is done in a set of training maps. In each map there are hundreds of episodes. After this training period, the Bot will play a set of test maps exploiting the acquired knowledge. The results are then observed. The behavior after the training period should be more efficient than that of the built-in AI. By trial and error the agent is expected to learn how to kite. It will decide when to attack and when to flee, making its units become more challenging targets.

1.1 Real-Time Strategy

Real-Time Strategy is a genre of video games which originated in the 1980s that became very popular in the 1990s. Today it has many adepts and there are many companies developing new games with important franchises like Company of Heroes™ and Total War™. Most of the games of this genre are published for PC. Generally, in an RTS game, two or more players compete in a map where they must gather resources to improve their economy, build bases and create armies to attack the adversaries and defend from their attacks. In this project, we'll focus only in the combat aspects of a match involving small groups of units. This is known as Micro-management.

1.2 Starcraft

StarCraft™ is an RTS game developed by Blizzard Entertainment and published on March 31, 1998 [2]. It's set in a fictitious future in a distant sector of the galaxy with three species fighting for dominance. Each of them with unique units, technologies, attributes and abilities: the Terrans, humans with very versatile and good defensive units; the Zerg, a race of insectoid aliens that can produce cheap, but rather weak, units in vast numbers; and the Protoss, an advanced alien species with very powerful but more expensive units. On November 30th 1998 an expansion pack was released, StarCraft: Brood War™, including new units and game updates. Although the three playable factions are very asymmetric, the game is very well balanced so that none of the factions have advantages over another. **Figure 1** shows a battle between a Protoss army and a Terran army.

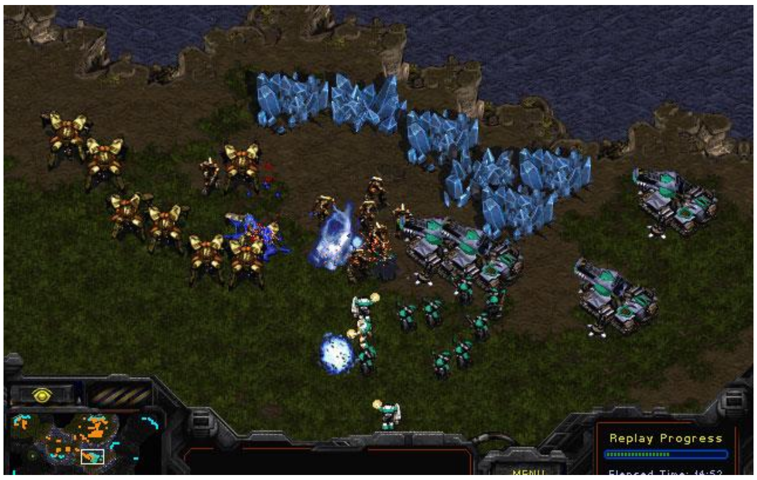


Figure 1. Combat between Protoss (left) and Terran (right) forces.

Starcraft has become a very extensive test-bed for AI research around the world. Many universities have developed Bots that compete against each other in international competitions like the Artificial Intelligence and Interactive Digital Entertainment (AIIDE) [3] and the Student Starcraft AI Tournament (SSCAIT) [4]. There is a very active community of developers and researchers working on many different approaches of AI.

2. The problem

RTS games involve many problems in AI. The environment is dynamic and stochastic, it constantly changes and actions have a probability of success. The information about the environment is incomplete: players only have information about the areas around their units. This is known as the 'fog of war'. High terrain is also not visible from a lower position.

Perhaps the most obvious problem is the very large state and action spaces of an RTS game's match. As estimated by Ontañon et al. [5], the number of possible states in an average map in Starcraft is many orders of magnitude bigger than the total states of board games like Chess or Go. The number of actions possible at each state is also hard to estimate due to the special abilities, different contexts and the internal state of a unit (health, weapons cool down, paralyzed, etc.), to tackle this problem some abstraction is necessary to reduce the states and actions space.

3. State of the art:

Reinforcement Learning has been successfully applied in RTS games, not only in Starcraft but in other games like Wargus [6] or Glest [7].

The Q-Learning algorithm along with the SARSA algorithm were studied in a combat scenario in Starcraft by Wender and Watson [8]. They compared the one-step version of both algorithms and their more sophisticated versions using eligibility traces.

Q-Learning was also studied by Mestres [9] in a combat scenario in Starcraft. He applied the algorithm simultaneously at two levels: squad and unit level. It allowed the units to perform complex actions that require units to work together (e.g., surrounding an enemy unit).

Both Wender and Watson [8] and Mestres [9] did their experiments in small custom maps against the built-in AI, which had a simple and highly deterministic behavior. When the built-in AI engages in a fight, it selects a target (generally the closest enemy unit), walks towards the target until it is inside the weapon's range and finally, attacks while the unit remains inside the weapon's range or until it has been destroyed. If the enemy unit leaves the weapon's range, the unit controlled by the built-in AI will chase it. A kiting unit can exploit that behavior by fleeing repeatedly and only stopping to attack. This technique is particularly efficient when it is used by range units against melee units (i.e., units with no range weapons).

A full StarCraft Bot generally mixes more than one AI technique assigned to different tasks, such as economy management, placing buildings, exploration and so on. A good example of a full Bot is the UAlbertaBot [10] by Churchill, which separates tasks into modules that are handled by different managers.

4. Motivation and Goal of this project

The aim of this project is to test which of these two approaches would be more suitable for the micro-management of range units on a full Starcraft Bot: the simple one-step Q-Learning or the Approximate Q-Learning using linear function approximation. The final result is measured in terms of total accumulated reward, time required to learn and win-rate.

4.1 The reinforcement learning problem:

In both versions of the algorithm, like in any other RL problem, the agent will interact with the environment in order to achieve a goal.

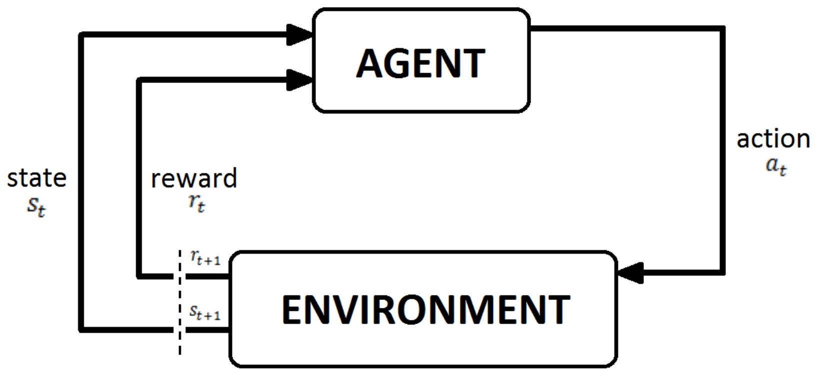


Figure 2. Interaction between agent and environment in RL.

As seen in **Figure 2**, at every time step $t = 1, 2, 3, \ldots$ the agent lands in a state $s_t \in S$, where S is the set of all possible states. From this state the agent takes an action $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ is the set of actions available at state s_t. As a result, the agent receives an immediate reward $r_{t+1} \in \mathcal{R}$ and lands in a new state $s_{t+1} \in S$. The agent's goal is to maximize the total reward received along all the time steps.

Although RTS games are perceived by human players to be Real-Time, they have discrete time steps. In Starcraft, for example, the game state changes at an average of 24 Frames per Second (hereafter, FPS). It means that the state of the game is updated approximately every 42 milliseconds. An important difference between Starcraft and simple RL problems is that in Starcraft some actions take more than one time step to finish. Even the same action, for example firing a range weapon, might take longer for some units than for others.

4.2 Markov Decision Processes

The definition of the task that the agent has to solve isn't trivial. To guarantee successful learning, the signals perceived by the agent at a given time step must retain the important information necessary to make its decisions. If all the relevant information from previous states is retained in the current state signal it's said to have *the Markov Property* [11]. In such case, the next state s_{t+1} and the immediate reward obtained r_{t+1} depend only on the current state s_t and the current action a_t, not on all the sequence of states and actions prior to step t.

$$\Pr\,\{\, s_{t+1} = s', r_{t+1} = r' \mid\, s_t, a_t \,\},$$

An RL task that satisfies the *Markov property* is said to be a Markov Decision Process (hereafter, MDP). An MDP is defined by the set of states S, the set of actions $\mathcal{A}$, the transition probabilities $\mathcal{P}^a_{s\,s'}$ (the probability to land from state s to state s' taking action a) and the expected value of the next immediate reward $\mathcal{R}^a_{s\,s'}$.

4.3 Definition of a state

The state signal is key because that is the only information that the agent will perceive and use in order to make its decisions. On the other hand, it's not possible to keep all the information available from the game because the state space would be too large. That would require an enormous amount of memory space for storage and the time necessary to explore each state would increase exponentially. Some abstraction is necessary. This is why the representation of the state is a trade-off between simplifying the states to reduce the state space and avoid missing valuable information.

In this project, the state happens at the unit level, meaning that each of the units owned by the Bot have their own current state. During the development of the project, some information about the state of a unit was dismissed and some information was added based on the agent's behavior and performance in the tests. The reason to use thresholds to categorize the values instead of using the real values is to keep the state space to a reasonable size.

The state information kept in the final version is:

- **HP**: hit points of our unit. The possible values are {High, Normal, Low, Critical}.
- **HP_ENEMIES**: sum of the hit points of all the enemies around our unit. The possible values are {High, Normal, Low, Critical}.
- **DPS:** the *damage per second* that our unit can inflict, calculated as:

$$dps(unit) = \frac{weapon\ damage\ \times frames\ per\ second}{weapon\ cool\ down}$$

  The possible values are {High, Normal, Low, Critical}.

- **DPS_ENEMIES:** the sum of the damage per second that the enemies around our unit can inflict, calculated as:

$$dps_{enemies} = \sum_{i=0}^{n} dps(\ enemy_i)$$

  The possible values are {High, Normal, Low, Critical}.

- **SPEED**: the unit's top speed. The possible values are {Fast, Medium, Slow}.
- **DISTANCE**: the distance between our unit and the closest enemy unit. The possible values are {Far, Medium, Close}.
- **COOLDOWN**: whether the unit's weapon is ready to shoot or not. The possible values are {Yes, No}.

- **ATTACKED**: whether the unit is under attack or not. The possible values are {Yes, No}.

As a result, the total size of the space state is $S = 4^4 \times 3^2 \times 2^2 = 9216$.

4.4 Possible actions

Something similar to the state space problem happens with the action space. The game gives many possible actions to each unit at a given moment. They vary depending on the context, the unit type or its internal state at that time. These actions need to be explored several times for each state to let the Q-Learning algorithm converge effectively to the optimal policy.

To keep things simple, only three actions are possible for each unit. These actions are not always available depending on the context of the unit at a given time. The possible actions are:

- **Explore**: this is the only action available when there are no enemies around. The purpose of this action is to encourage the unit to find the enemies.

- **Attack**: this action, along with *'Flee'*, are the only actions possible when there are enemies around the unit. It calls upon the Attack Manager that decides which enemy to attack.

- **Flee**: this action, along with *'Attack'*, are the only actions possible when there are enemies around the unit. It calls upon the Flee Manager that decides where to flee.

A further explanation of what these actions do is found later in section **6.4**.

4.5 Rewards

The last main component of the MDP are the rewards. A reward is a signal, it comes as a number perceived from the environment by the agent at each time step $r_t \in \mathfrak{R}$. The goal of an RL agent is to maximize the accumulated rewards in the long run. Rewards can also be negative, in which case are usually referred to as penalties.

Like defining the state, the rewards were modified, added and removed while testing the agent. These are the rewards in the final version and their values:

- **Win (100)**: is received when the match ends in victory for our RL agent.

- **Lose (-100)**: is received when the match ends in defeat for our RL agent.

- **Alive (-0.01):** is received every time step until the match ends, its purpose is to encourage the agent to finish the match as soon as possible.

- **Enemy Destroyed (50)**: when an enemy unit is destroyed, this reward is received by the unit who destroyed it.

- **Unit Lost (-20)**: is received when our unit is destroyed.

- **Unit Damaged (-5)**: is received when our unit has been hit since the last time step but has not been destroyed yet.

The values are not necessarily optimal and can be modified depending on the desired priorities (e.g., finishing the match fast vs. surviving as long as possible).

4.6 Q-Learning

The Q-Learning algorithm was introduced by Watkins in 1989 [11]. It's an off-policy TD control algorithm. Off-policy means it is independent from the policy being followed, with enough exploration (i.e., all the pairs continue to be updated) it's guaranteed to converge to $Q*$, the optimal action-value function. A policy, denoted π, is a map that shows the agent which action to take at any state. Temporal-Difference (hereafter, TD) means that it approximates its current estimate based on previous estimates without waiting for a final outcome (also known as bootstrapping).

Q-Learning uses state-action pairs and calculates the 'quality' of these pairs, named Q-Values. That is, the expected return of taking action a from state s and thereafter following the optimal policy π. The simple one-step update formula for Q-Learning is:

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

A Q-Learning agent is not a reflex agent, meaning, it doesn't have any prior knowledge about the environment, it doesn't have a policy, it doesn't know the transition function nor the reward function. It has to learn by trial and error.

To keep the number of possible actions very short, the agent doesn't need to learn who to attack, where to flee or how to explore the terrain. It just needs to learn when to attack, flee or explore, and the manager modules transform this decision to specific enemies or positions on the map. The managers are described in section **6.4**.

4.7 Approximate Q-Learning (features based Q-Learning)

In some cases when the state space is too large, it becomes impossible to keep a table with all the Q-Values. It's also impossible to explore every state-action pair enough times to guarantee that the Q-Learning algorithm will converge to the optimal policy. The algorithm treats every pair as unique and there's no relation between pairs regardless of how similar they might seem. The idea of the Approximate Q-Learning is to generalize over all this similar states-action pairs and use this knowledge when the agent lands in a new state that has never been explored. This means that the learning becomes faster, since the agent uses the knowledge from the visited states to make decisions in the new ones.

To generalize over state-action pairs some relevant features must be extracted, these features are real numbers (e.g., number of enemies around the unit). Similarly to the state representation in exact Q-Learning, selecting the features that represent a state-action pair is key because this is the only information that the algorithm will use to learn. The result of extracting these features is a vector of real numbers (to simplify things, all features are integer numbers in this project).

The goal is to give these features a weight, so that we have a weights vector with the same length as the features vector. After every time step, when a feature is present ($feature\_value \neq 0$) the weight of this feature is updated. They are a measure of how good or bad that feature is, if the rewards are high, the features that lead to that reward are increased, if the rewards are negative, the features are decreased.

Instead of updating the Q-Values, like in exact Q-Learning, only the weights vector is updated.

$$w_i \leftarrow w_i + \alpha[\left(r + \gamma \max_{a'} Q(s', a')\right) - Q(s,a)] \times f_i(s,a)$$

The Q-Value of a state-action pair is calculated as the dot product of the features vector and the weights vector.

$$Q(s,a) \leftarrow w_0 f_0(s,a) + w_1 f_1(s,a) + \cdots + w_n f_n(s,a)$$

The features are extracted from state-action pairs, which means they have information not only about the state but also about the possible outcome of the action taken. Those features are generally binary, their value is 1 if they are present and 0 otherwise. These are the features defined for this project:

These features are extracted from the current state:

- $\boldsymbol{f_0(s,a)}$: the *Hit Points* of our unit.
- $\boldsymbol{f_1(s,a)}$: the sum of *Hit Points* of all the enemies around our unit.
- $\boldsymbol{f_2(s,a)}$: the *Damage per Second* that our unit can inflict.
- $\boldsymbol{f_3(s,a)}$: the *Damage per Second* of all the enemies around our unit.
- $\boldsymbol{f_4(s,a)}$: the speed of our unit.
- $\boldsymbol{f_5(s,a)}$: the distance to the closest enemy.
- $\boldsymbol{f_6(s,a)}$: the number of enemies around the unit.
- $\boldsymbol{f_7(s,a)}$: the number of allied units around our unit.
- $\boldsymbol{f_8(s,a)}$: the unit's weapon *cool down* frames.
- $\boldsymbol{f_9(s,a)}$: whether the unit is under attack or not.

These features are assumptions based on the current state and the action taken:

- $\boldsymbol{f_{10}(s,a)}$: action $\boldsymbol{a}$ takes our unit most likely closer to an enemy.
- $\boldsymbol{f_{11}(s,a)}$: action $\boldsymbol{a}$ takes our unit most likely away from an enemy.
- $\boldsymbol{f_{12}(s,a)}$: action $\boldsymbol{a}$ will most likely hurt an enemy.
- $\boldsymbol{f_{13}(s,a)}$: action $\boldsymbol{a}$ will most likely destroy an enemy.
- $\boldsymbol{f_{14}(s,a)}$: action $\boldsymbol{a}$ will most likely cause us damage from an enemy.
- $\boldsymbol{f_{15}(s,a)}$: action $\boldsymbol{a}$ will most likely cause us death from an enemy.
- $\boldsymbol{f_{16}(s,a)}$: action $\boldsymbol{a}$ takes our unit most likely closer to an enemy.

5. BWAPI

The Brood War Application Programming Interface (hereafter, BWAPI) is an API for interacting with *Stacraft: Broodwar* (1.16.1) [12]. It is a free and open source C++ framework that allows developers to interact with the game and control its units like a human player would do. The cheating options, such as human input or complete map visibility, are disabled by default so that the AI agents have to work under partial information conditions.

The Brood War Application Programming has become very popular for AI researchers and AI developers as it is well documented and has a very active community. A list of academic works using BWAPI can be found in [13]. This Project has been developed using BWAPI 4.1.1 and MS Visual studio 2013. The Project is written in the C++ programming language.

Our RL agent's goal is to solve a problem in form of an MDP. This is a brief description of some of the most relevant aspects of BWAPI used in this project. It helps to show how the MDP can be modeled using this framework.

5.1 Interfaces

5.1.1 **AIModule:** is the main class that initializes BWAPI. It has to be implemented or inherited by a custom class. Its member functions are used to handle the game events. A description of the events is found in section **5.2**.

5.1.2 **Game:** contains all the information available from the game state, including units, players, terrain, fog of war, regions, etc.

5.1.3 **Player:** represents a player in the match, every player has a *Player* instance. This interface allows us to get all the information about our own units and control them (when player = *self*) or to get all the information available about the enemy (when player = *enemy*).

5.1.4 **Unit:** is used to issue commands to individual units, like attack or move, and provides all the information about a specific unit, like ID, exists, hit points, unit type, position, etc.

5.2 Events

There are many events that the BWAPI extracts from the game. The **AIModule** catches these events and they can be handled writing their respective callback methods. All the events should be handled in a full *Starcraft* Bot to benefit from their information. Here, only the events used in this project are described. For a list of all the events, please check the BWAPI's class diagram available in [14].

5.2.1 **OnStart:** this event is triggered only once at the beginning of the match. It's not possible to keep information in memory from one match to the next so this event is ideal to load files from disk (e.g., the table containing the Q-Values).

5.2.2 OnEnd: this event is triggered only once at the end of the match. It is ideal to store information to disk. In our MDP, this event represents the final state and its only possible action is *EXIT*. It has a Boolean parameter *isWinner* which is *true* when our agent has won the match and *false* otherwise. This event is very useful to give our agent a final reward.

5.2.3 OnFrame: this happens every game frame. It could be considered as a time step in our MDP but further considerations need to be done because not all actions require the same time to be finished. Treating every frame as a time step has another complication explained in section **5.3**.

5.2.4 OnUnitDestroyed: this event is triggered every time a unit is destroyed, regardless of who its owner was. The *Unit* interface can then be used to know the former owner. It is useful to return positive rewards when the unit was an enemy and negative rewards when our agent lost a unit.

5.3 Frames

Usually, the game runs at 24 frames per second, which means that the state of the game changes approximately every 42 milliseconds. It seems reasonable to treat every game frame as a time step of our MDP but there is another important concept to take into account, the *Latency Frames*.

The *Latency Frames* are the number of frames before a command can be processed. Issuing commands to soon $LatencyFrames < (NextComandFrame - lastCommandFrame)$ would result in undesired behavior of the units (e.g., units get blocked and are unable to move or attack), this is known as 'spamming commands'. This problem can be solved by simply skipping the frames in between.

$$if\ (\ FrameCount\ \%\ LatencyFrames \neq 0)\ return;$$

5.4 Tiles & Pixels

The map coordinates can be represented in pixels (Position) or in tiles (TilePosition) which are squares of 32x32 pixels. The pixels are very useful to calculate the exact distance between two positions or to measure a unit's speed, to name a few. They are not very good representatives of a unit's state, though. The average size of a map in *Starcraft: Broodwar* is 128 x 128 pixels. It means that there are approximately 16384 different positions where a unit can be at a given time. This information isn't very practical, at least not in exact Q-Learning, because there are too many different values. Using tiles could be a way to reduce the possible values. Section **6.2** describes the implementation of a simple MDP using BWAPI. The state is represented as the *x* and *y* coordinates (in tiles) of a unit. As described in section **6.4.1**, the flee manager also uses tiles, named blocks, to calculate an influence map around the unit and pick the safest block to escape.

6. Implementation

6.1 Serialization

It’s not possible for BWAPI to keep data in memory from one match to another. However, it’s possible to load this data from a file at the beginning of a match and save it back to the disk when the match ends. This is done respectively when the events **onStart** and **onEnd** are triggered.

The Q-Values are the only information relevant to the agent that needs to be stored. A Q-Value is just a real number linked to a state-action pair as explained in section **4.6**. A map seemed like a good choice for this task. The *Boost C++ Libraries* [15] contains many useful data structures including unordered maps that accept custom classes as key. So, the Q-Values are stored in a *unordered_map*: the key is a custom class that contains the state-action pair and the value is simply the Q-Value of this pair. The map is created during the first match and then serialized and saved to the disk when the match ends, ready to be loaded at the beginning of the next match.

6.2 Gridworld

This first experiment was made to test the viability of the Q-Learning algorithm using BWAPI. As shown in **Figure 3**. the agent has to learn the path to a goal state where a big positive reward awaits (100), the other final states (the blue areas north, south and west) cause the agent to lose and receive a negative reward (-100). Apart from the final rewards, there is a living penalty (-0.5) every time step that the agent makes a transition to a non-final state. It’s a small negative reward that encourages the agent to finish the task as quickly as possible.

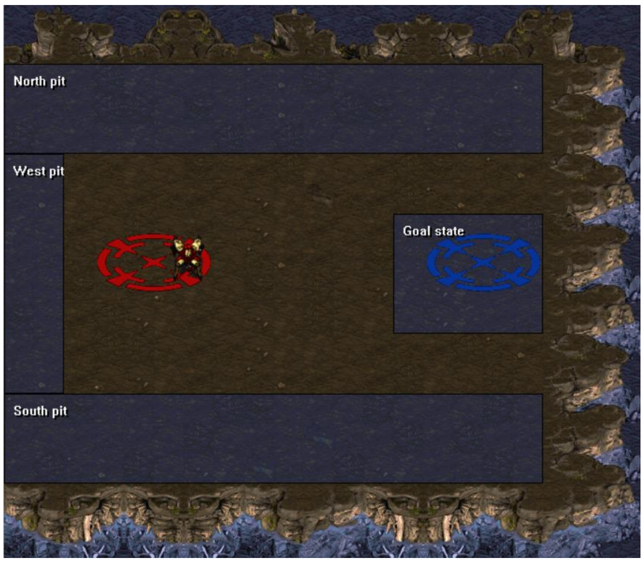
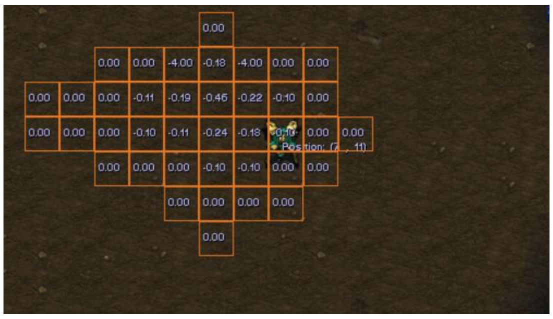


Figure 3. Gridworld experiment.

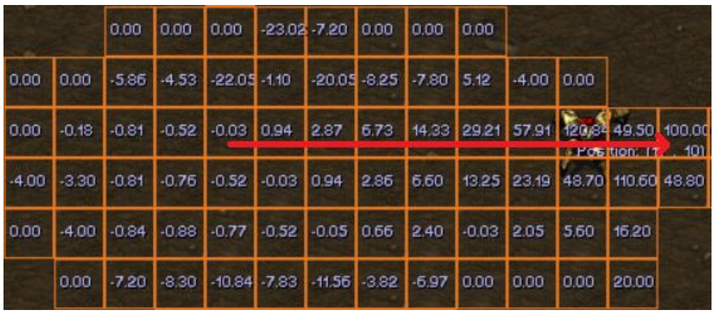
In this simple task the state of our only unit can be represented by its position on the map. Using tiles, instead of pixels, the state space is reduced to approximately 60 different states. As the agent updates the Q-Values, the one with the highest value is displayed on the map; that's the value of the state-action pair that the greedy agent would choose.



					0.00				
		0.00	0.00	-4.00	-0.18	-4.00	0.00	0.00	
0.00	0.00	0.00	-0.11	-0.19	-0.46	-0.22	-0.10	0.00	
0.00	0.00	0.00	-0.10	-0.11	-0.24	-0.18	-0.10	0.00	0.00
		0.00	0.00	0.00	-0.10	-0.10	0.00	0.00	
				0.00	0.00	0.00	0.00		
					0.00				

Figure 4. Exploration, the highest Q-Value of each state is displayed in the grid.

After 500 episodes with an exploration rate $\varepsilon = 0.4$ (meaning that 40% of the time the agent acts greedy and 60% randomly) the Q-Values have propagated enough throughout the map. Assuming that the values are optimal, the agent can now follow an optimal policy π_* by setting $\varepsilon = 1$ (acting greedy), the agent becomes a reflex agent and exploits the knowledge from the learning episodes. The experiment worked as planned and proved that Q-Learning can be applied in Starcraft using BWAPI. The implementation of this experiment can be used as a starting point for more complex MDPs.



		0.00	0.00	0.00	-23.02	-7.20	0.00	0.00	0.00				
0.00	0.00	-5.86	-4.53	-22.05	-1.10	-20.05	-8.25	-7.80	5.12	-4.00	0.00		
0.00	-0.18	-0.81	-0.52	-0.03	0.94	2.87	6.73	14.33	29.21	57.91	120.84	49.50	100.00
-4.00	-3.30	-0.81	-0.76	-0.52	-0.03	0.94	2.86	6.60	13.25	23.19	48.70	110.60	48.80
0.00	-4.00	-0.84	-0.88	-0.77	-0.52	-0.05	0.66	2.40	-0.03	2.05	5.60	16.20	
	0.00	-7.20	-8.30	-10.84	-7.83	-11.56	-3.82	-6.97	0.00	0.00	0.00	20.00	

Figure 5. Following the optimal policy π_* in the grildworld.

6.3 Kiting

Kiting is a technique used in RTS games' micro-management. It consists of attacking and fleeing repeatedly in order to avoid being exposed to the enemy weapons for too long. It's particularly useful when it's used by fast range units against slow melee units. If it's well performed, a single unit could deal with a much bigger force and remain unharmed.

The definition of the states, actions and rewards of an MDP to learn how to kite in Starcraft have been explained in sections **4.3**, **4.4** and **4.5**. Here, we can see the resultant class diagram of the two different implementations of a Kiting Bot and a short description of their classes.

6.3.1 Exact Q-Learning

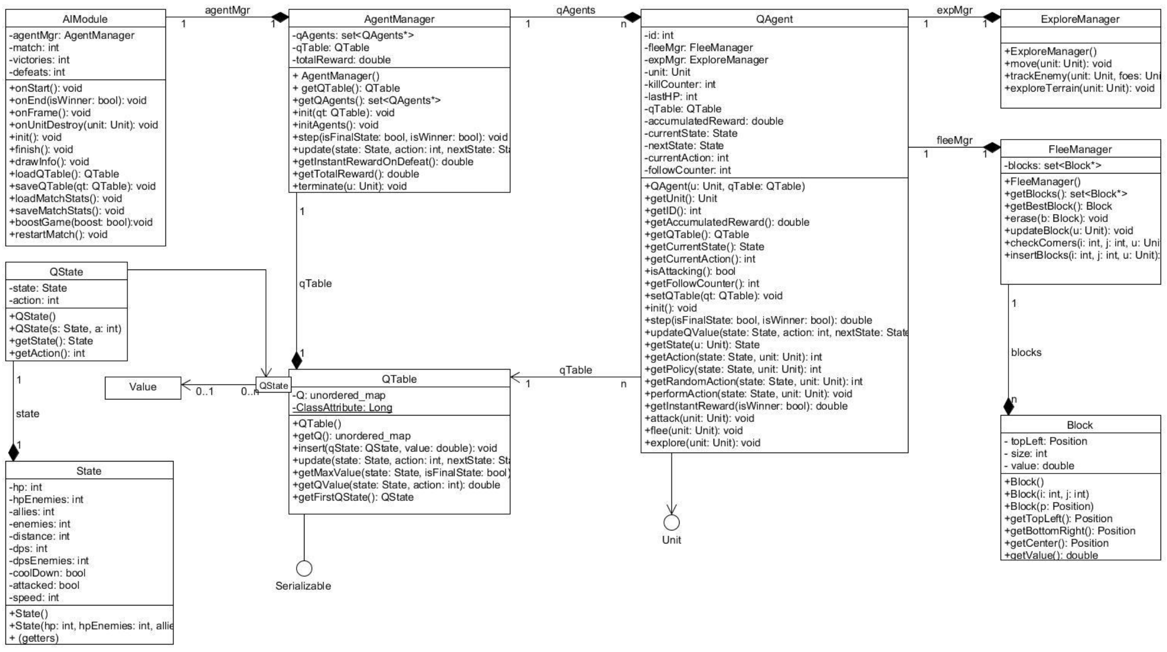


Figure 6. Class diagram of the Q-Learning Kiting Bot.

6.3.1.1. AIModule: inherits from *BWAPI::AIModule*, this is the main class of the Bot. It handles the events, loads and saves the information from disk and initializes the *AgentManager*.

6.3.1.2. AgentManager: is the owner of the *QTable*, a custom class that contains the map with all the Q-Values. It initializes a set of agents associated to each existing own unit in the game.

6.3.1.3. QAgent: is associated to a unit in the game, it has a reference to the *QTable* passed by its parent class *AgentManager*. It interprets the unit's state, takes actions and updates the Q-Value of the last state-action pair visited. Note: due to the simplicity of the attack method this class also includes the instructions of the *AttackManager*, there's no separate manager module for the attack.

6.3.1.4. QTable: is a custom class that contains the unordered map with all the Q-Values. The keys are the visited state-action pairs and the values are the current estimated value of that state-action pair, the values are initialized to 0. It provides the *QAgent* class with methods to update, insert and access the Q-Values in the map.

6.3.1.5. QState: represents a state-action pair.

6.3.1.6. State: is the representation of the current state of a unit in the game.

6.3.1.7. FleeManager: is called by the *QAgent* class when the 'flee' action is taken. It executes the instructions to flee from the enemy units. Section **6.4.1** has more details about how the agent flees.

6.3.1.8. ExploreManager: is called by the *QAgent* class when the 'explore' action is taken. It executes the instructions to perform the exploration of the terrain. Section **6.4.3** has more details about the way the exploration is performed.

6.3.1.9. Block: is a class used by the *FleeManager* class to create a grid around the unit and estimate which cell, or block, is the safest at a given time.

6.3.2. Approximate Q-Learning

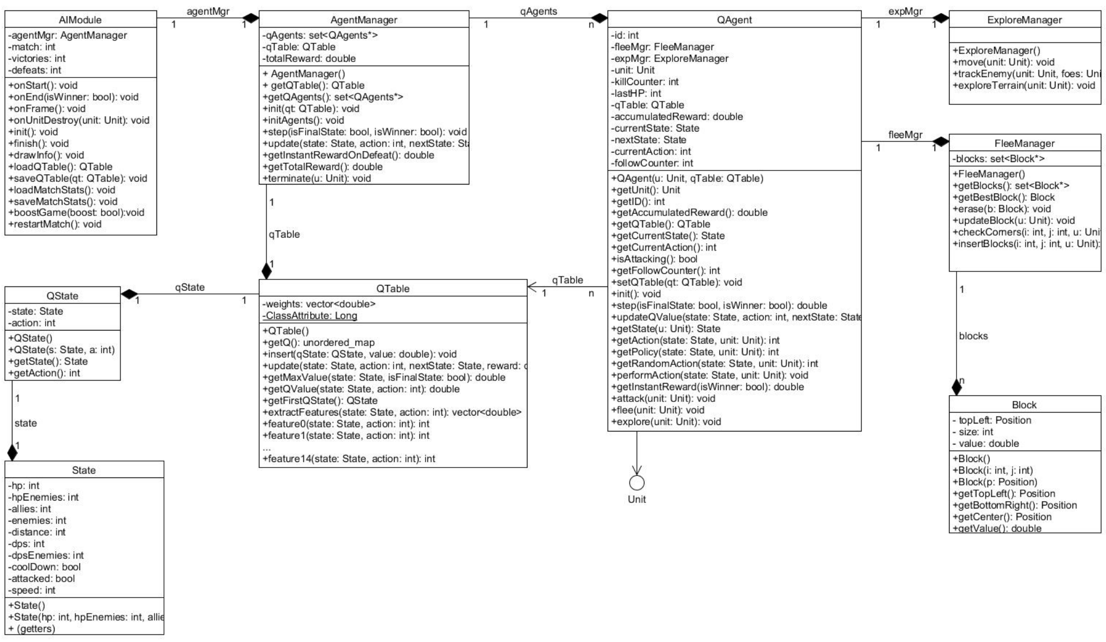

AIModule
-agentMgr: AgentManager
-match: int
-victories: int
-defeats: int
+onStart(): void
+onEnd(isWinner: bool): void
+onFrame(): void
+onUnitDestroy(unit: Unit): void
+init(): void
+finish(): void
+drawInfo(): void
+loadQTable(): QTable
+saveQTable(qt: QTable): void
+loadMatchStats(): void
+saveMatchStats(): void
+boostGame(boost: bool):void
+restartMatch(): void
agentMgr
1
1
AgentManager
-qAgents: set<QAgents*>
-qTable: QTable
-totalReward: double
+ AgentManager()
+ getQTable(): QTable
+getQAgents(): set<QAgents*>
+init(qt: QTable): void
+initAgents(): void
+step(isFinalState: bool, isWinner: bool): void
+update(state: State, action: int, nextState: St
+getInstantRewardOnDefeat(): double
+getTotalReward(): double
+terminate(u: Unit): void
qAgents
1
n
QAgent
-id: int
-fleeMgr: FleeManager
-expMgr: ExploreManager
-unit: Unit
-killCounter: int
-lastHP: int
-qTable: QTable
-accumulatedReward: double
-currentState: State
-nextState: State
-currentAction: int
-followCounter: int
+QAgent(u: Unit, qTable: QTable)
+getUnit(): Unit
+getID(): int
+getAccumulatedReward(): double
+getQTable(): QTable
+getCurrentState(): State
+getCurrentAction(): int
+isAttacking(): bool
+getFollowCounter(): int
+setQTable(qt: QTable): void
+init(): void
+step(isFinalState: bool, isWinner: bool): double
+updateQValue(state: State, action: int, nextState: State
+getState(u: Unit): State
+getAction(state: State, unit: Unit): int
+getPolicy(state: State, unit: Unit): int
+getRandomAction(state: State, unit: Unit): int
+performAction(state: State, unit: Unit): void
+getInstantReward(isWinner: bool): double
+attack(unit: Unit): void
+flee(unit: Unit): void
+explore(unit: Unit): void
expMgr
1
1
ExploreManager
+ExploreManager()
+move(unit: Unit): void
+trackEnemy(unit: Unit, foes: Uni
+exploreTerrain(unit: Unit): void
fleeMgr
1
1
FleeManager
-blocks: set<Block*>
+FleeManager()
+getBlocks(): set<Block*>
+getBestBlock(): Block
+erase(b: Block): void
+updateBlock(u: Unit): void
+checkCorners(i: int, j: int, u: Uni
+insertBlocks(i: int, j: int, u: Unit):
1
blocks
n
Block
- topLeft: Position
- size: int
- value: double
+Block()
+Block(i: int, j: int)
+Block(p: Position)
+getTopLeft(): Position
+getBottomRight(): Position
+getCenter(): Position
+getValue(): double
1
qTable
1
qState
1
1
QState
-state: State
-action: int
+QState()
+QState(s: State, a: int)
+getState(): State
+getAction(): int
QTable
-weights: vector<double>
-ClassAttribute: Long
+QTable()
+getQ(): unordered_map
+insert(qState: QState, value: double): void
+update(state: State, action: int, nextState: State, reward: d
+getMaxValue(state: State, isFinalState: bool): double
+getQValue(state: State, action: int): double
+getFirstQState(): QState
+extractFeatures(state: State, action: int): vector<double>
+feature0(state: State, action: int): int
+feature1(state: State, action: int): int
...
+feature14(state: State, action: int): int
qTable
1
n
1
state
1
State
-hp: int
-hpEnemies: int
-allies: int
-enemies: int
-distance: int
-dps: int
-dpsEnemies: int
-coolDown: bool
-attacked: bool
-speed: int
+State()
+State(hp: int, hpEnemies: int, allie
+ (getters)
Unit

Figure 7. Class diagram of the Approximate Q-Learning Kiting Bot**.**

6.3.2.1. AIModule: instead of loading the unordered map, it loads the weights vector at the beginning of the match and saves it to a file at the end.

6.3.2.2. QTable: no longer has a map of all the Q-Values visited. Instead it has methods to extract the features from a *State*. It also keeps the vector with the weights of the features and methods to update them, as explained in section **4.7**.

All the other classes are the same as in the exact Q-Learning implementation in section **6.3.1**.

6.4 Managers

The managers are the code executed when the RL agent wants to perform an action (i.e., attack, flee and explore). Their purpose is to release the RL agent from the details of those actions, so that the state space of the MDP stays small.

6.4.1 Flee Manager

This module is called by the agents when the action *Flee* is selected. Initially the idea was to use a simple fleeing method. The method originally tested was to run from the enemy whose position was the closest to our unit. This method proved to perform very poorly and the unit usually ended up running straight to another enemy or getting itself trapped between the enemy and obstacles.

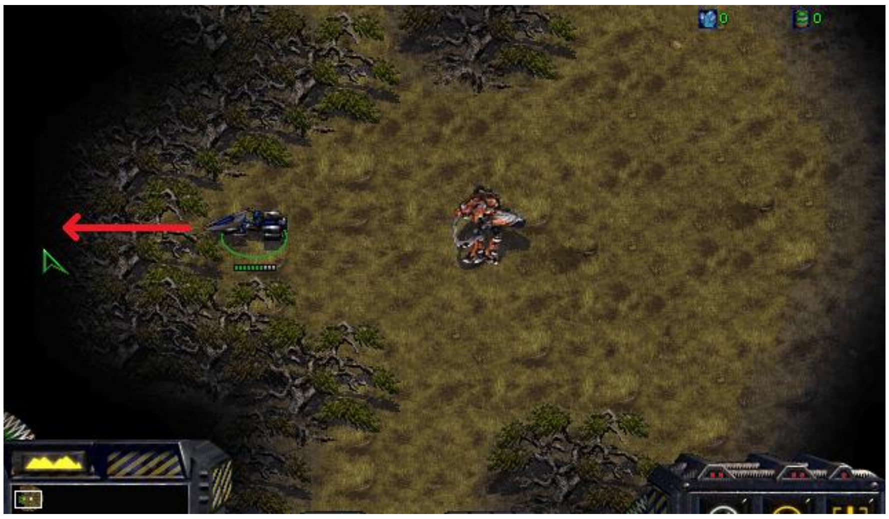


Figure 8. Fail case of fleeing from the enemy to the opposite direction.

The Next approach tested was to calculate the mean point of all the enemy positions near the agent, then run away from this point. There are many counter examples where this solution won't work as expected because the information about the current position of the enemy units is lost, many enemy distributions can compute the same mean point and be completely different. Finally, the solution that proved to perform better than the others and still remain simple in concept and computational cost was using Influence Maps to calculate the safest position around the agent in a given state. At every time step the safest position is updated while the agent continues to flee.

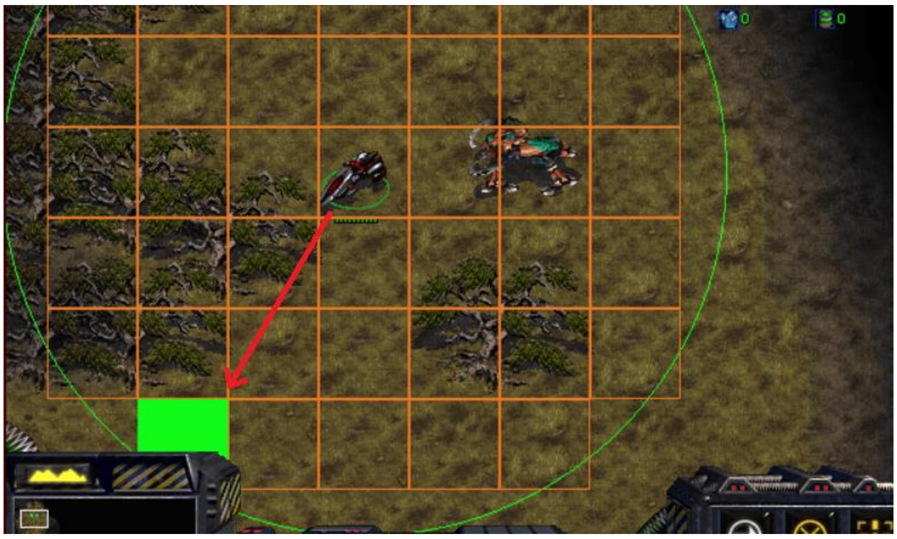


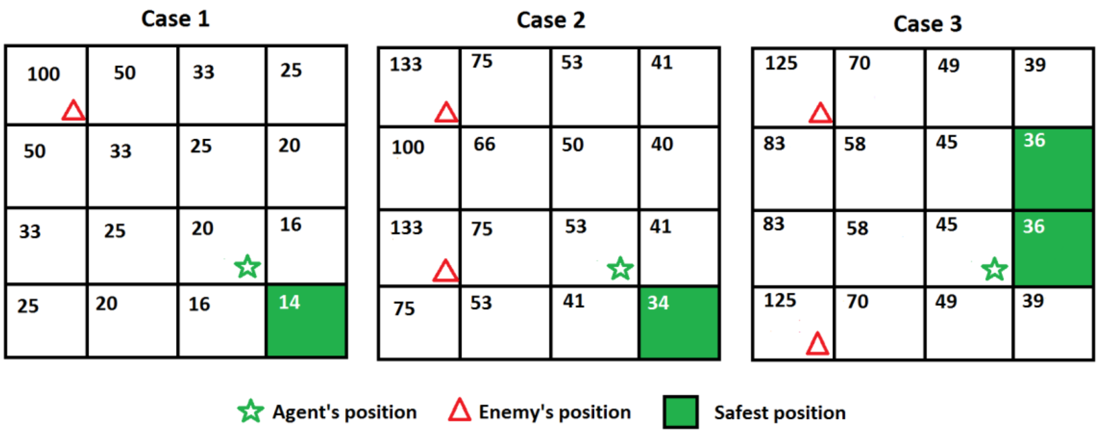
Figure 9. Vulture fleeing using an influence map.

The value v of a cell tells the agent how dangerous that cell is. The higher the value, the more dangerous it is. It's computed using the formula:

$$v(c) = \sum_{i=0}^{n}(100/Manhattan(c,i) + 1).$$

Where n are the enemies in the grid and $Manhattan(c,i)$ is the distance from a cell c to enemy i. The agent chooses the cell with the lowest v.

$$flee\_cell = \underset{c}{\arg\min}(v(c)).$$



Case 1

100 △	50	33	25
50	33	25	20
33	25	20 ☆	16
25	20	16	14 🟩

Case 2

133 △	75	53	41
100	66	50	40
133 △	75	53 ☆	41
75	53	41	34 🟩

Case 3

125 △	70	49	39
83	58	45	36 🟩
83	58	45 ☆	36 🟩
125 △	70	49	39

☆ Agent's position △ Enemy's position 🟩 Safest position

Figure 10. Three examples of the safest current position using influence maps.

6.4.2 Attack Manager

Deciding which enemy to attack when there are many options is another topic that has been widely researched. In this project two greedy solutions are used to keep things simple. The first greedy method tested was attacking the closest enemy. This method minimizes the distance that the unit has to walk if the enemy is out of the weapon's maximum range; however, when mixed with kiting technics the unit can change its target, spreading the inflicted damage among the enemies.

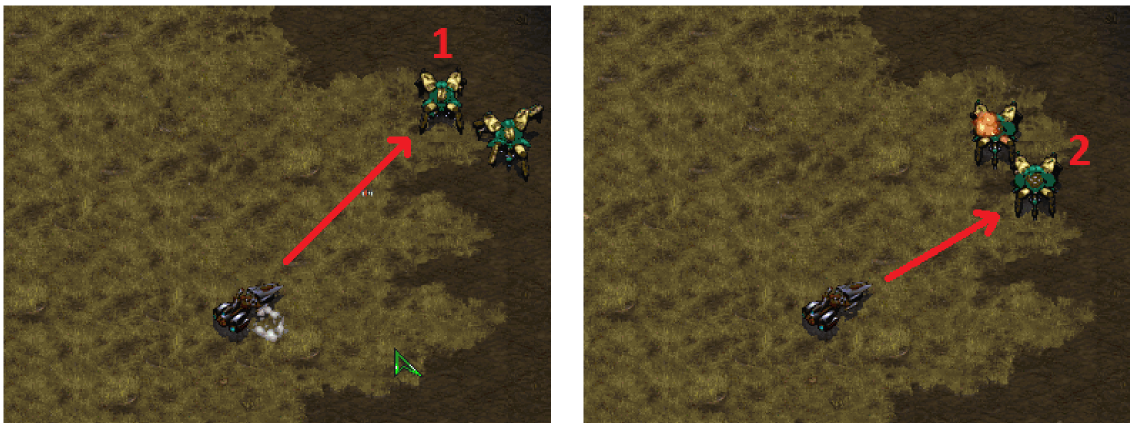


Figure 11. Vulture switching target as the second Dragoon gets closer.

The second greedy method tested was to attack the weakest enemy. This method focuses all of our agents' fire power in destroying the enemy with least *hit Points*, thus, reducing the enemy's number. The drawback of this method is that our agents become blind and focus only in reaching the weakest target, even if it means exposing itself to other enemy units. Another problem with this method is known as 'over killing' an enemy, when all the units focus their fire power on an enemy that could be easily destroyed with fewer units.

One method with effective and simple results was mixing these two previous methods. Using the reciprocal value of the closest enemy's distance and the reciprocal value of the weakest enemy's hit points, each value weighted 0.5. The agent picks the enemy unit i with the highest value $v(i)$.

$$target = \arg\max_i(v(i));$$

$$v(i) = \left(0.5 * \frac{1}{dist(i)} + 0.5 * \frac{1}{hp(i)}\right)$$

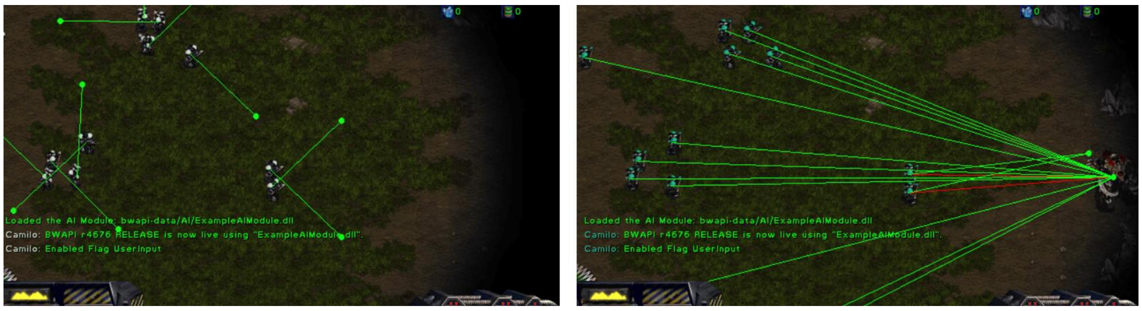
6.4.3 Explore Manager

This module is called by our agents when the action *Explore* is selected. The purpose of this module was not to develop efficient exploration of the map. Given the lack of complete information in RTS games, exploration itself is a very wide problem and it is beyond the purpose of this project. This module aims to give the agent some behavior to find the enemies and therefore prevent the match from playing indefinitely.

To explore the map our agents will follow a random pattern while no enemies are spotted. If the agent changes direction every time step the movement becomes very erratic and the agent generally stays in a small area. Always going straight would be very inefficient after facing any obstacle or the edge of the map. Mixing these two actions turns into a very basic and relatively efficient exploration method. Specifically, the agents change direction randomly 20%

of the time steps and continue straight the other 80%. These percentages were picked after testing different values.

What happens when one unit doesn't see any enemy but another unit does? In this case the exploration manager module is also called but the behavior is very different. As seen in the **Figure 12**, the module uses the built-in path-finding algorithm to go to the location where the closest enemy was spotted, producing a *call effect* to all the agents that run to help their ally.



```
Loaded the AI Module: bwapi-data/AI/ExampleAIModule.dll
Camilo: BWAPI r4676 RELEASE is now live using "ExampleAIModule.dll".
Camilo: Enabled Flag UserInput
```

```
Loaded the AI Module: bwapi-data/AI/ExampleAIModule.dll
Camilo: BWAPI r4676 RELEASE is now live using "ExampleAIModule.dll".
Camilo: Enabled Flag UserInput
```

Figure 12. Two Marines find the Ultralisk, the other Marines head to where it was seen.

7. Results

The agents were trained in 8 different maps during 1200 episodes. The training was divided into groups of 200 episodes (25 episodes on each map). After each of these groups, the Bots played four test maps (courtesy of Alberto Uriarte). In these four maps the Bots played 100 episodes each map exploiting the knowledge from the training part. The maps are shown in sections **7.1** to **7.4**. The following graphs show the win-rate of both Bots (exact and approximate Q-Learning) in the testing maps.

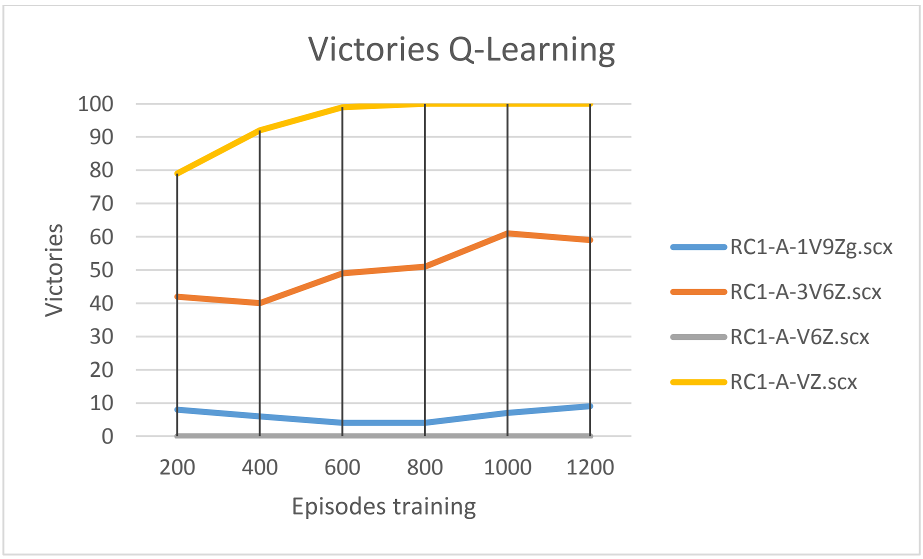


Figure 13. Q-Learning win-rate

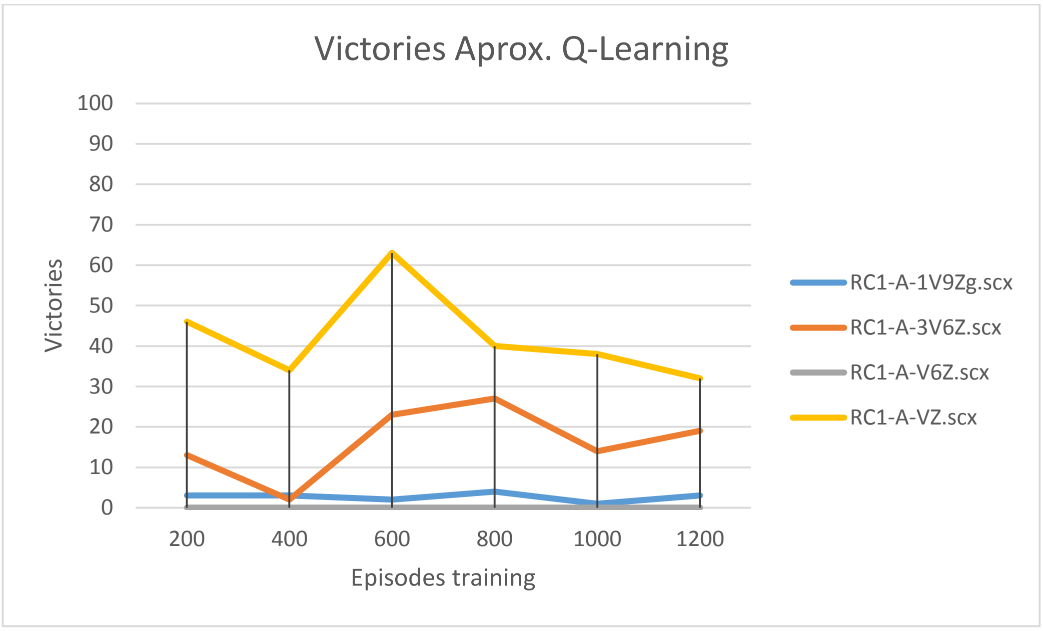


Figure 14. Approximate Q-Learning win-rate.

In **Figure 13** it is visible that the Q-Learning Bot improved its win-rate in two of the maps while it continued to fail in the other two. The Approximate Q-Learning Bot (**Figure 14**) also improved at first in those two maps but then worsened towards the last part of the test. The maps *RC1-A-1V9Zg.scx* and *RC1-A-V6Z.scx* are still beyond the Bots' capabilities. Only a few victories were achieved in *RC1-A-1V9Zg.scx* while both Bots failed 100% of the time in map *RC1-A-V6Z.scx*. It is worth mentioning that the win-rate of the built-in AI in these maps was always 0%.

Apart from the win-rate, the total accumulated reward achieved during each episode was recorded to a file and displayed using histograms in sections from **7.1** to **7.4**. The horizontal axis represents the reward's values and the vertical axis represents the frequency of these values.

7.1 Map RC1-A-VZ.scx

This map is a small battleground surrounded by a wall. Our Bot controls one Vulture (fast range unit), the enemy controls a Zealot (melee unit).

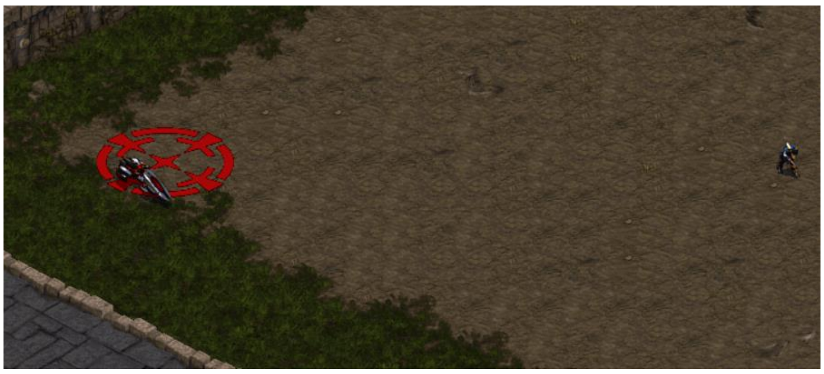


Figure 15. testing map RC1-A-VZ.scx

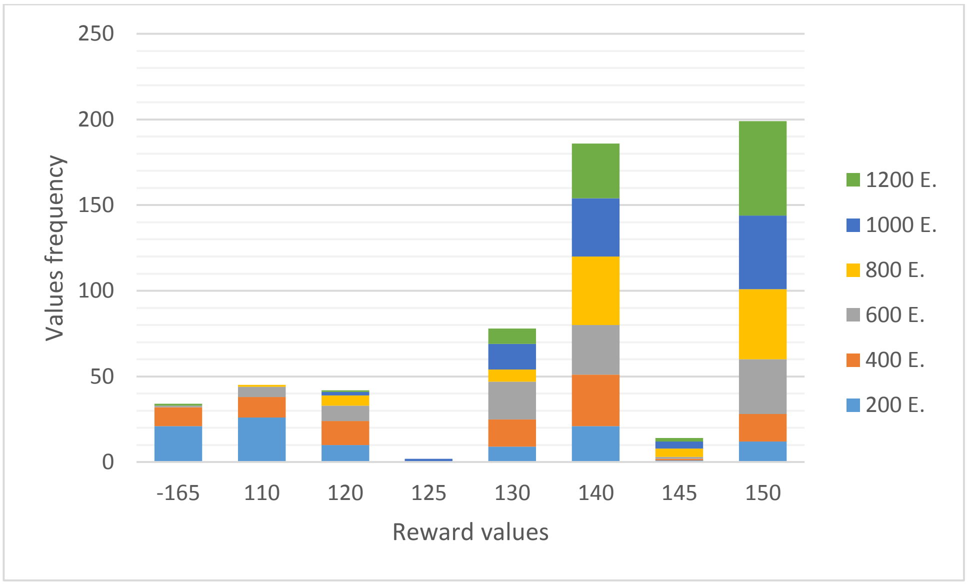


Figure 16. Q-Learning, histogram of map RC1-A-VZ.scx

The results show that the accumulated rewards were higher for the Bot with more training. The Bot collected higher rewards after training 1200 episodes (green) and 1000 episodes (dark blue). Only the first bar shows negative rewards meaning that the Bot lost on that episode. This happens mostly when the Bot had only trained during 200 (light blue) and 400 (orange) episodes.

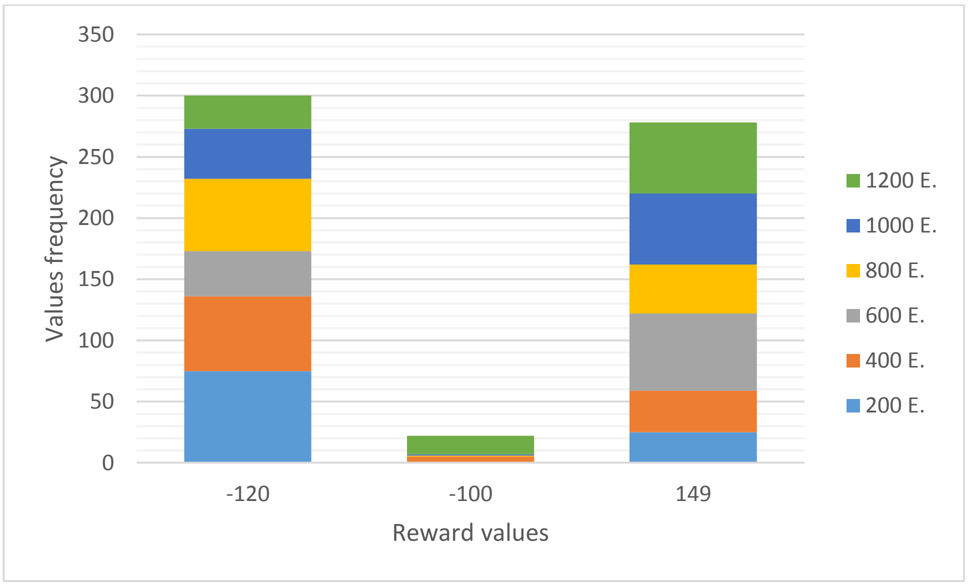


Figure 17. Approximate Q-Learning, histogram of map RC1-A-VZ.scx

Similarly to the Q-Learning counterpart, the Bot improved the accumulated rewards as the training episodes increased. Something interesting is that the episodes ended faster than the Q-Learning Bot and only three different values of accumulated rewards were obtained.

7.2 Map RC1-A-V6Z.scx

This map is a small battleground surrounded by a wall. Our Bot controls one Vulture (fast range unit), the enemy controls six Zealots (melee units).

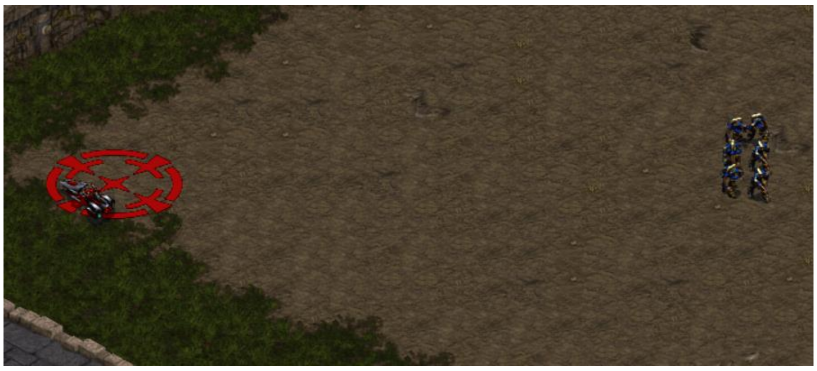


Figure 18. testing map RC1-A-V6Z.scx

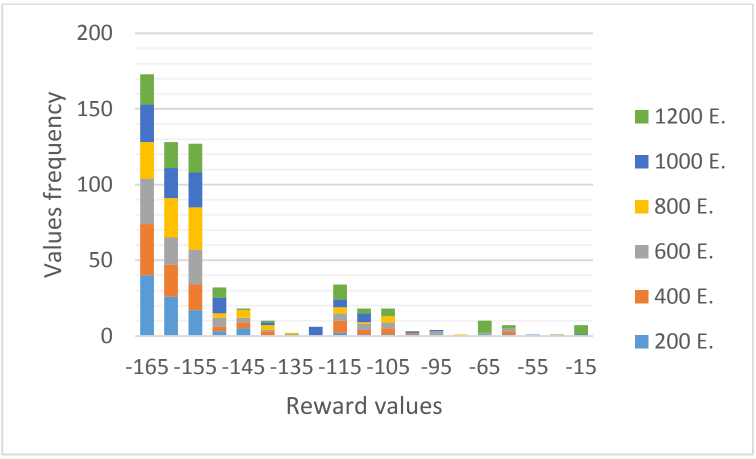


Figure 19. Q-Learning, histogram of map RC1-A-V6Z.scx

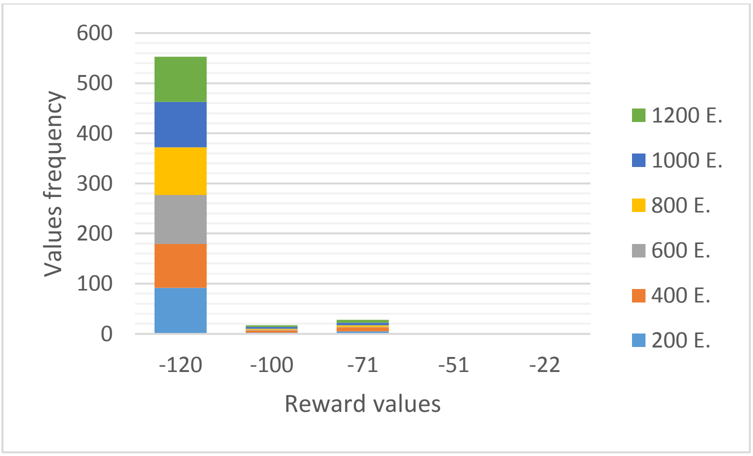


Figure 20. Approximate Q-Learning, histogram of map RC1-A-V6Z.scx

In this map, both Bots, Q-Learning and approximate Q-Learning, failed to achieve victory in all the episodes. It's not unexpected considering the superiority of the enemy force. Nevertheless, an optimal Kiting behavior should be able to take advantage of the Vulture's speed and range weapon to defeat the six Zealots unharmed.

7.3 Map RC1-A-3V6Z.scx

This map is a small battleground surrounded by a wall. Our Bot controls three Vultures (fast range units), the enemy controls six Zealots (melee units).

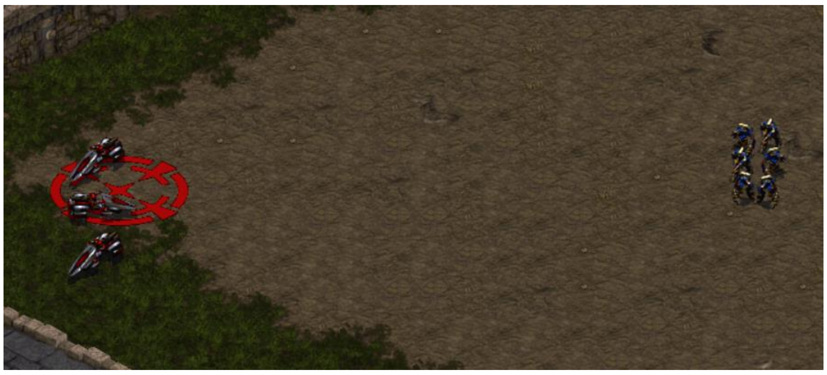


Figure 21. testing map RC1-A-3V6Z.scx

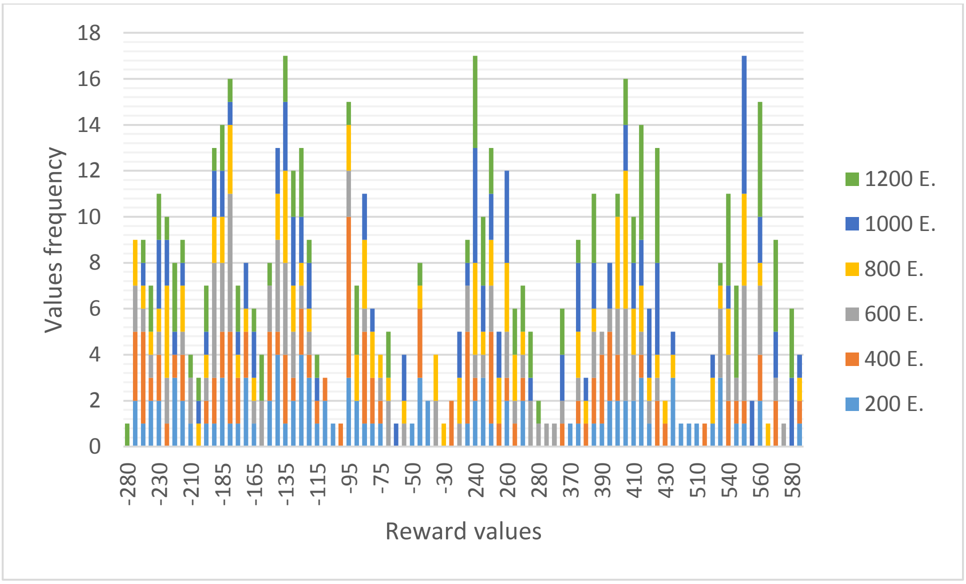


Figure 22. Q-Learning, histogram of map RC1-A-3V6Z.scx

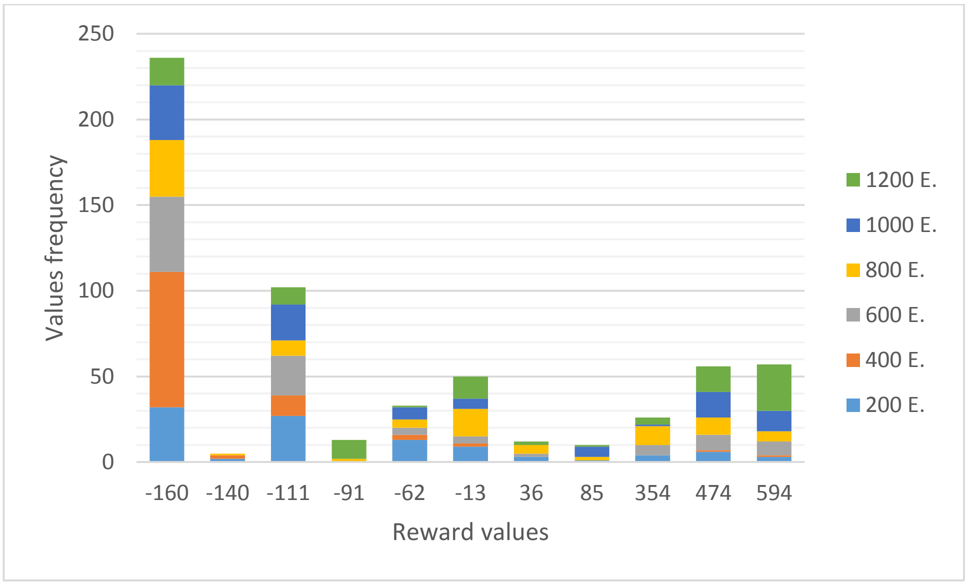


Figure 23. Approximate Q-Learning, histogram of map RC1-A-3V6Z.scx

The map configuration is the same as in section **7.2** adding two more Vultures to our force. This maps shows that a small force of three Vultures can achieve victory roughly half of the time. It is not optimal but it is much better than the results from one single Vulture in **7.2**. Like in the previous maps, the frequency of rewards received by the Q-Learning Bot is more diverse than the rewards received by the approximate Q-Learning Bot.

7.4 Map RC1-A-1V9Zg.scx

This map is a small battleground surrounded by a wall. Our Bot controls one Vulture (fast range unit), the enemy controls nine Zerlings (fast melee units).

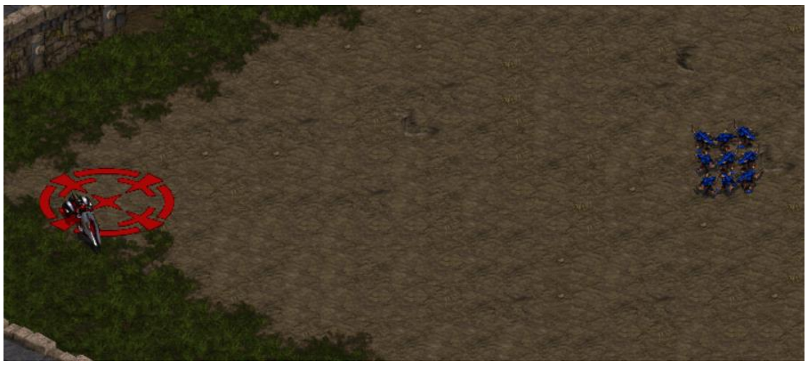


Figure 24. testing map RC1-A-1V9Zg.scx

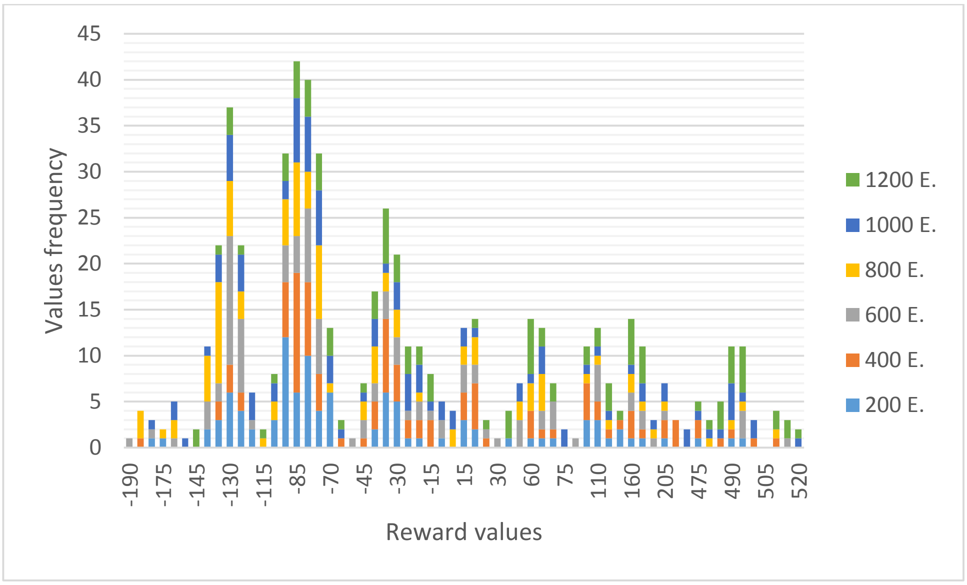


Figure 25. Q-Learning, histogram of map RC1-A-1V9Zg.scx

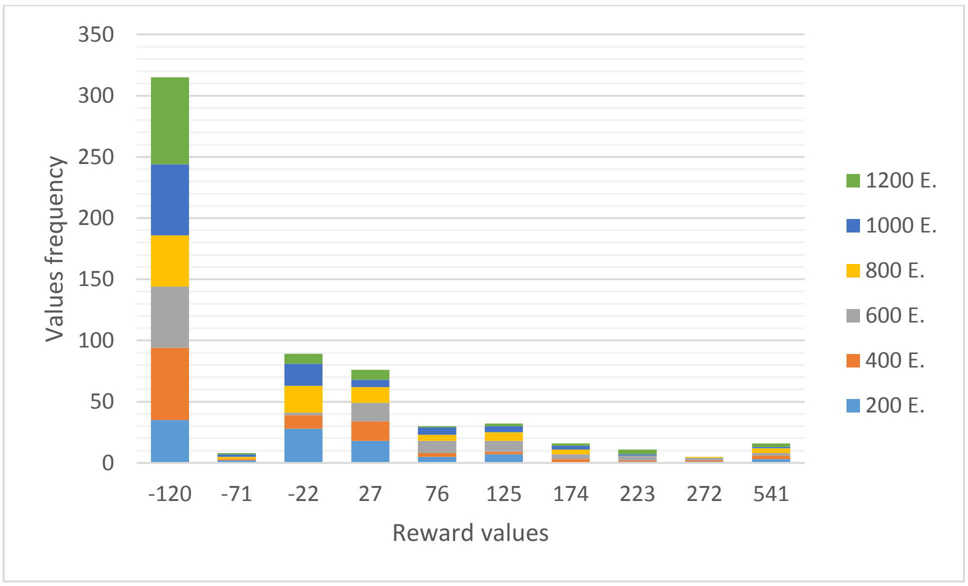


Figure 26. Approximate Q-Learning, histogram of map RC1-A-1V9Zg.scx

The results from map *RC1-A-1V9Zg.scx* are very interesting because of the big number of enemies and their speed of movement. Here, a Vulture has to engage a force of nine Zerlings, weak but fast melee units. The win-rate in this map is very low for both Bots (under 10%). The histogram does not show a clear improvement between the low trained version of the Bots and the high trained versions. The frequency of rewards shown in the histograms are very diverse as well. With so many threats, the result is highly stochastic. The Vulture only won when it had 'good luck'.

8. Conclusions

Both approaches highly improved the behavior of the units in terms of combat skills comparing to the built-in AI. The built-in AI doesn't include Kiting as part of its micro-management. In that sense, both Bots learnt to attack when the conditions were good (i.e., an enemy is damaged and/or alone) and when to flee (i.e., outnumbered by the enemy force, when the weapon is not ready to shoot or when being under a heavy attack).

The approximate Q-Learning Bot proved to learn much faster than the Q-Learning Bot, usually reaching its peak of success after the first 200 episodes of training. This could be useful when the time designated to learn is short. In the other hand, this approach can result in unexpected behavior if the features are not chosen well, particularly those based on assumptions (see section **4.7**). The use of features can also result in the Bot 'de-learning' from state-action pairs that it already knew well. For example, if the Bot is good in one map where attacking is the best choice, then it is exposed to a long learning period in a very different map where fleeing is the only good choice, and finally tested back in the first map. The Bot might still think that fleeing is the best choice because the weight of the features have changed.

The Q-Learning Bot proved to not be as good as its approximate counterpart when the training period was short, but it improved considerably after many episodes of training. Unlike the approximate Q-learning approach, the Q-Learning Bot never forgot a good action from a particular state because its Q-Value is stored in its internal table.

As mentioned in section **7**, neither of the Bots performed the Kiting in an optimal way. One of the reasons for that is that the fleeing manager, the code in charge of moving the unit away from the enemy, is not optimal. That code is not part of the Q-Learning implementation but it is critical to Kite. The next section talks about the future work, including how the fleeing manager could be improved.

The definition of a state can always be improved to achieve better results. The code was made very modular so that it could be done by modifying only the State class. Unfortunately, if the state definition changes, the previous serialized files containing the agent's knowledge become invalid. Every time the State definition changes, the learning process has to start from scratch.

Finally, this rather small project proves that RL techniques can be applied to RTS games' AI and videogames in general. It only requires that companies be willing to dedicate more resources to bring this to big commercial games.

9. Future work

Potential Fields: using this method instead of Influence Maps, the unit could compute a safe route instead of a safe point to flee, as shown by Uriarte and Ontañon [16].

Angle of attack: this problem appeared during the kiting test with Vultures. In some occasions the vulture selected a target located behind. It had to stop, turn around, shoot and accelerate again. Sometimes that gave other enemies necessary time to reach and attack our unit. This didn't happen when it selected a target in front, with a margin of about 20°. In that case the Vulture could shoot while escaping at top speed. A Bot that keeps that information into account could perform a better kiting.

Overkill: as mentioned before in section **6.4.2**, this problem appears when too many of our units attack a weak or damaged enemy unit simultaneously. In a battle this means that all these units have to wait for their weapons to cool down before they can attack again, making them vulnerable to other enemy units. Solving this problem would prevent the Bot from wasting fire power unnecessarily and thus, focusing in other threats.

Add to an existing Bot: including the achievements of this project in an existing Bot with simple micro-management should improve its efficiency.

Explore Macro-Management: it would be interesting to explore the capabilities of RL in other areas of RTS games, like placing buildings or resource management.

10. References

[1] BURO, M. 2004. Call for AI research in RTS games. In Proceedings of the AAAI Workshop on AI in Games, AAAI Press, 139–141.

[2] "StarCraft's 10-Year Anniversary: A Retrospective," Blizzard Entertainment, Retrieved 2008-03-31. [Online] Available: http://www.blizzard.com/us/press/10-years-starcraft.html.

[3] AIIDE Starcraft AI Competition, http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/

[4] SSCAIT, Student StarCraft AI Tournament, http://sscaitournament.com/

[5] S. Ontañon et al., "A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft," IEEE Transactions on Computational Intelligence and AI in games, IEEE Computational Intelligence Society, 2013.

[6] U. Jaidee and H. Muñoz-Avila, "CLASSQ-L: A Q-Learning Algorithm for Adversarial Real-Time Strategy Games," Artificial Intelligence in Adversarial Real-Time Games: Papers from the 2012 AIIDE Workshop AAAI Technical Report WS-12-15.

[7] V. K. Dimitriadis, "Reinforcement Learning in Real Time Strategy Games Case Study on the Free Software Game Glest,"

[8] S. Wender and I. Watson, "Applying reinforcement learning to small scale combat in the real-time strategy game starcraft:broodwar," in CIG, 2012.

[9] F. Mestres, "Q-Learning in an Open-Space Combat Scenario for Real-Time Strategy Games," Master's thesis, Universitat Autònoma de Barcelona, 2014.

[10] David Churchill, "UAlbertaBot – Starcraft AI Competition Bot", Available: https://github.com/davechurchill/ualbertabot

[11] R. Sutton and A. Barto, "Reinforcement Learning: An Introduction," MIT Press, 1998.

[12] BWAPI: An API for interacting with Starcfraft: Broodwar (1.16.1) Available: http://bwapi.github.io/

[13] BWAPI Academic papers, [Online] Available: https://github.com/bwapi/bwapi/wiki/Academics

[14] BWAPI Overview, [Online] Available: https://code.google.com/p/bwapi/wiki/BWAPIOverview

[15] Boost C++ libraries, Available: http://www.boost.org/doc/libs/1_58_0/more/getting_started/index.html

[16] A. Uriarte and S. Ontañon, "Kiting in RTS Games Using Influence Maps", Artificial Intelligence in Adversarial Real-Time Games: Papers from the 2012 AIIDE Workshop AAAI Technical Report WS-12-15.