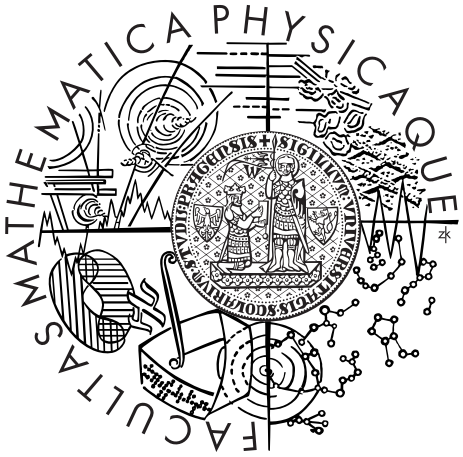
Charles University in Prague
Faculty of Mathematics and Physics

**BACHELOR THESIS**




Tomáš Witzany

# Adaptive Agent in a FPS Game

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Rufolf Kadlec
Study programme: Computer Science
Specialization: Programming

Prague 2013

**Dedication**

I would like to thank my supervisor Mgr. Rudolf Kadlec for all his input and help. I would also like to thank all the people behind Pogamut for enabling so many people to write AI. And last but not least I would like to thank my family for supporting me as much as they could.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date .....  signature of the author

Název práce: Adaptivní agent v FPS hře

Autor: Tomáš Witzany

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Rudolf Kadlec

Abstrakt: V této práci je navržen a implementován adaptivní protihráč v počítačové hře Unreal Tournament v jejím módu Deathmatch. Agent byl navržen pomocí zpětnovazebního učení a implementován na platformě Pogamut. Pro stavovou abstrakci byl použit clusterovací algoritmus k-means. Dále byl na platformě Pogamut vyvinut framework pro testování výkonu agentů. Tento framework byl použit pro provedení množství experimentů testující různé strategie pro výběr akcí a také byly otestovány různé parametry Q-Learning algoritmu. Výsledné chování má výkon srovnatelný s implementacemi zpětnovazebního učení popsanými v dostupné literatuře.

Klíčová slova: Adaptivní agent, FPS, Pogamut, zpětnovazební učení

Title: Adaptive Agent in a FPS Game

Author: Tomáš Witzany

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Rudolf Kadlec

Abstract: In this work I design and implement an adaptive oponent for the computer game Unreal Tournament for its Deathmatch mode. The agent has been designed using reinforcement learning and implemented on the Pogamut platform. A k-means clustering algorithm has been used for state abstraction. Furthermore an agent performance testing framework has been developed for the Pogamut platform aswell and used in this work. Several experiments testing different action-selection policies and different parameters of the Q-Learning algorithm were conducted. The resulting behaviour has a performance comparative to other implementations of reinforcement learning from other literature.

Keywords: Adaptive agent, FPS, Pogamut, reinforcement learning

# Contents

# 1. Introduction

Artificial Intelligence (AI) is used in many types of different computer programs but games have always been a popular test-bed for artificial intelligence research [4, 7, 12, 16]. What we see today is an ever increasing amount of AI in computer games aswell. And since current computer games are becoming more and more realistic there is much demand for the AI in games to become realistic aswell.

There are several video games that use advanced AI techniques to achieve realism such as Black and White's emergent AI creature [29] or the simulated life in S.T.A.L.K.E.R. [9]. However the need for game AI to be fun and very strictly controlled makes use of very simple AI solutions often preferred over complicated solutions that cost a lot of time to develop [32]. Even then computer games are perfect for simulating realistic environments and are often used for research in real-time environment AI [15, 21, 22].

This work explores the possibilities of reinforcement learning in a first-person shooter (FPS) and its main goal is the design, implementation and performance measurement of an adaptive opponent in the computer game Unreal Tournament (UT) in its mode Deathmatch (DM).

In these games the player interacts with a real-life, or close to real-life simulation from the viewpoint of his character and his goal is usually to eliminate his opponents. The player controlled characters in these games are called avatars and characters controlled by the computer are called bots. FPS are very often played over the Internet with more human players but bots can replace the human players when necessary. However FPS bots are not usually very competitive against human opponents.

Several platforms were used to implement the agent. To control the bot I used Pogamut [14], which is a Java middleware for programming bots in UT. Pogamut is a Gamebots [18] wrapper, which is a text protocol used for bot programming. Furthermore I developed a plugin used for automatic bot testing in Pogamut as my year project [31]. Since I'm trying to make a realistic opponent, I decided he will have to adapt to his environment and therefore I needed to use an online learning algorithm. I used Q-learning [10] because it is learning an action-value function which I will use to control higher level functions of the bot.

To get a good idea of the AIs performance I decided to use the same sensory information as HunterBot uses. HunterBot is a rule based bot provided with the Pogamut installation as an example bot. The resulting AI has about one third of the performance of HunterBot. As the HunterBot's state space was too complex for the RL algorithm to perform well I used clustering to reduce the state-space of the original HunterBot. Additionally several experiments were made with different action selection policies and different combinations of learning and discount rates were tested aswell.

## 1.1 Thesis structure

The thesis is divided into eight chapters. The first chapter is an introduction and the second chapter are works related to Q-Learning, reinforcement learning or AI in FPS games in general.

The third chapter describes the theoretical background of the implementation. Chapter 3.1 describes reinforcement learning in general and Chapter 3.2 focuses solely on Temporal-Difference learning and Q-Learning.

The fourth chapter is about third party tools, frameworks or software I used in the implementation. Chapter 4.1 describes Unreal Tournament, its environment and the game rules. Chapter 4.2 describes the Pogamut platform and Chapter 4.3 describes the testing plugin I developed to test the bots automatically.

The fifth chapter concerns itself with the implementation issues and the architecture of the bot. Chapter 5.1 describes the implementation architecture, Chapter 5.2 describes how the learning part of the bot is implemented and Chapter 5.4 describes how and why I use a clustering algorithm.

The sixth chapter is the results and discussion about their performance. The seventh chapter writes about possible future direction of this agent and then the last chapter concludes the thesis in Chapter 8.

# 2. Related work

Most current bots that use some kind of reinforcement learning [26] can be divided into one of these categories. The first being a bot, where the policy learned controls primitive functions of the agent, such as a certain part of his body and the sensory input is similarly primitive, such as ray-casting information. The second type uses reinforcement learning to teach the bot a higher level decision process. For example how to select from a set of hard coded actions or optimizing some sort of strategy, such as selecting what navigation point to go to and leaving the actual navigation for the developer to implement. Some works presented here combine the two approaches and learn both a higher level decision process and then the primitive functions aswell.

In the following chapter I describe the approaches in several different papers and compare them to the methods used in this work.

- Michelle McPartland and Marcus Gallagher compare several RL approaches in their paper [22]. They combine both of the two approaches mentioned and develop both high and low level AI in a hierarchical RL model. The bot learns to decide whether to use a combat or navigation controller and both of these controller are evolved separately. Except their hierarchicalRL bot, they develop a complete RL model aswell, where the bot learns the entire combat and navigation task using a flat RL structure. An eligibility trace version of the Sarsa algorithm called Sarsa($\lambda$) [26] is used as the learning algorithm.

  The results of this comparison show that a hierarchical model learns faster and performs better than the flat RL, but even then it is much worse than a state machine AI. The performance achieved by the hierarchical model is 5 average kills per match compared to 14.1 of the state machine AI. The flat RL model performs even worse with 1.9 frags on average. In my bot I use a similar approach in that I use several hard coded actions and the bot learns which one to take. However i dont learn the primitive functions of the bot as these are hard-coded.

- In [21] Megan Smith, Stephen Lee-Urban and Héctor Muñoz-Avila present an on-line RL algorithm that teaches a team of bots a winning policy in the Dominate game-mode in UT. Dominate is a game-mode where a team of bots wins by accumulating points that the team gets from capturing and holding control points. The learning algorithm is a one-step Q-Learning [26] algorithm and it uses bots with fixed behavior. The policy learned tells the bots what control point to go to capture and hold.

  One of the advantages of this approach is that the learning algorithm works independently on how the bots are programmed and that the state space in this model is very small and therefore the policy evolves very fast. The resulting policy performed very well even against sophisticated strategies and even dynamically changing strategies. The RL algorithm was able to adapt to dynamically changing strategies usually within the span of a few matches. They tested their AI against HTNbots, which is a system that dynamically changes its strategies, and achieved a performance almost the

same as HTNbots. The similarity to my work is that a high-level decision process is evolved and the rest of the behavior is hard-coded. However the enviroment and goal of the AI is completely different from mine.

- In [15] Ahmed S. Hefny, Ayat A. Hatem, Mahmoud M. Shalaby and Amir F. Atiya present Cerberus, a RL framework for CTF games. It uses single-step Q-Learning to select high level team actions and neural networks to learn fighting and movement behavior offline. The resulting performance of this framework was comparable with hard-coded agents. Again the similarity is that Cerberus uses RL to teach high-level reasoning, while the low-level functions have a predefined, fixed behavior even if they are evolved offline. The difference is that the high-level reasoning is designed for CTF actions, which are very different than the AI I made. The results presented in this work do not compare the performance to any other AI, instead observations of the AI's behavior are presented.

- In [25] B. Subagdja, W. Wang, A.-H. Tan, Y.-S. Tan and L.-N. Teow study the effect of a episodic and semantic memory on reinforcement learning. They develop a memory model to be used in learning to improve weapon selection of a FPS bot for Unreal Tournament, Deathmatch mode. The bot they use for testing is a bot very similar to HunterBot which I use in my work and they use a modified version that has its weapon selection strategy replaced by their learning implementation. By learning better weapon selection the bots performance increases, they also present results showing that the memory model increased the performance over a simple reinforcement learning bot.

- In [8] Remco Bonse, Ward Kockelkorn, Ruben Smelik, Pim Veelders and Wilco Moerman implemented a Q-Learning subsystem for a hard-coded bot in Quake III. They used Q-Learning to learn better performing combat movement of a bot, that was otherwise implemented by hand. The bot did not learn on-line and the performance achieved was about a 15% increase in kills per match.

# 3. Methods used

This chapter describes the methods and algorithms used in the implementation of my bots behavior. The first section provides a general overview of reinforcement learning and how it differs from supervised learning. The second section describes how temporal-difference learning works and shows us the Sarsa and Q-Learning algorithms that can be used to implement reinforcement learning.

## 3.1 Reinforcement learning

Reinforcement learning (RL) is an area of machine learning that teaches an agent how to take actions in a certain environment to maximize his reward [24] [26]. The main difference from standard supervised learning is that there are no predefined correct action-state pairs. Instead these must be discovered as actions that yield the biggest reward. One of the important features of RL is taking delayed reward into account. Delayed reward is the fact that an action may not only affect the immediate reward, but also the rewards in future situations.

RL is best used in environments that are interactive or where the agent has to be able to learn from its own experience. Standard supervised learning is learning from examples provided by a external supervisor. This is not very effective when we need to learn from interaction, because often it is very hard to find correct and representative examples of all the situations the agent can appear in.

The basic reinforcement learning model consists of an state space $S$, action space $A$ and a reward function $r : S \times A \to \Re$. Additionally sometimes rules restricting the actions in certain states are necessary, for example an agent cannot shoot when he does not have ammo. The action space $A$ is a set of actions the agent can perform in the environment. The state space $S$ is a set of n-tuples of variables describing the agents state, in our case for example his health or ammo, and sensory information such as visibility of certain points of interest.

In reinforcement learning a reward function defines the goal of the reinforcement learning problem. It is a function $r : S \times A \to \Re$ mapping the state of the environment or a state-action pair to a number — the reward. The value of $r$ represents immediate rewards for our actions. The RL agents objective is to maximize the total reward it receives and not the immediate. A key feature of the reward function is that it is unalterable by the agent, because it defines the features of the problem the agent is facing. However, reward functions may be stochastic and therefore not constant. The reward function in concept replaces the supervisor from supervised learning. But instead of finding representative examples of correct behavior we can define the goal of the behavior, which is often much simpler.

When an RL agent performs an action $a$ in an environment which is in a certain state $s$, the environment returns a reward based on the user defined reward function $r(s, a)$. The reward is then used to reinforce the estimated outcome of the decision the RL agent made. This estimated outcome is a state-action pair mapping of how well we expect an action to perform in a state and it is called a policy. A policy is a function $\pi : S \times A \to \Re$ that maps an state or a state-action pair to an expected reward. This policy evolves over time and determines the

path the agent should take to maximize his reward for the task defined in the reward function $r$. The reward function $r$ indicates what is good in an immediate sense and the policy $\pi$ value specifies what is good in the long run. For example in a game of chess if we take a queen, the reward function $r$ would give us a very high reward, but if that move results in the opponent winning the game in a few moves, the policy function $\pi$ value in this action-state would be low.

One of the challenges that arise in reinforcement learning is the explore-exploit trade-off. To be able to get the highest reward, we have to enable the agent to explore the actions with the highest reward. However it has to also prefer (exploit) the actions that the agent tried in the past and that have high expected reward. The problem is that we cannot exclusively prefer exploration nor exploitation. The agent has to explore a variety of actions and with time progressively start exploiting those that have the highest expected reward. The explore-exploit dilemma is more discussed in Chapter 5.3.

## 3.2 Temporal-Difference learning

Temporal-Difference learning (TD) is a reinforcement learning technique combining the advantages of Monte Carlo [5] and dynamic programming [6]. TD takes several ideas from Monte Carlo methods. Same as Monte Carlo methods, TD learns from repeated randomized simulations and can learn from experience without a model of the environment. TD is similar to dynamic programming in that it develops an estimate over time. Same as dynamic programming, TD updates its current estimate based on previously learned estimates (bootstrapping).

The simplest TD method known as TD(0) is

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{3.1}$$

where $V$ is the estimate of the policy $\pi$ being evolved, $s_t \in S$ is the state in time $t \in \aleph$, $r_t \in \Re$ the reward in time $t$ and $\alpha \in [0, 1]$ and $\gamma \in [0, 1]$ are constant parameters. The parameter $\alpha$ is called the learning rate and it determines to what extent will the new experience override the old policy estimate. The second parameter $\gamma$ is called the discount factor and it determines the importance of future rewards. A low value would make the policy opportunistic and a high value will make the policy plan ahead. A value between 0 and 1 is necessary.

As TD is basically a combination of Monte Carlo and dynamic programming, there are many similarities between TD and the Monte-Carlo method. The main difference from a Monte Carlo method is that Monte Carlo methods have to wait until the end of the episode to update the policy. A very simple Monte Carlo method would be:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_{t+1} - V(s_t)] \tag{3.2}$$

where $R_t \in \Re$ is the actual return following time $t$. As we can see, in TD(0) we can evaluate the function immediately in the step following the decision.

As discussed in Chapter 3.1 we have to trade off exploration and exploitation, and the two main types of approaches are: on-policy and off-policy. An on-policy method learns the value of the policy being carried out by the agent and off-policy learning learns the optimal policy independently on the agents actions. Sarsa is a on-policy method and Q-Learning is a off-policy method. The one-step Sarsa

and Q-Learning algorithm [26] are both TD learning methods which we discuss in the following chapters.

### 3.2.1 Sarsa

As Sarsa is an on-policy method, we must estimate $Q^{\pi}(s,a) : S \times A \rightarrow \Re$ for the current policy $\pi$ and for all the action-state pairs $(s,a) \in S \times A$. This can be done using the same TD method described above as TD(0) because formally we can consider state-action pairs to be states in TD(0) and then we get:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)] \tag{3.3}$$

This update uses every element of the quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ which is the source for the algorithms name Sarsa. The algorithm is presented in pseudo-code in Listing 3.1.

Listing 3.1: Sarsa

```
initialize Q(s,a) arbitrarily
repeat for each episode:
  initialize s
  choose a from s using policy derived from Q
  repeat for each step of episode:
    take action a, observe r, s'
    choose a' from s' using policy derived from Q
    Q(s,a) = Q(s,a) + α[r + γQ(s',a') - Q(s,a)]
    s=s'; a=a'
  until s is terminal
```

### 3.2.2 Q-Learning

The off-policy TD learning algorithm known as Q-Learning is one of the most important algorithms in reinforcement learning. The TD variant and its most simplest form is as follows:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)] \tag{3.4}$$

The main difference from Sarsa is that in this case, the learnt action-value function directly approximates the optimal action-value function. Chris Watkins, Peter Dayan proofed Q-Learning's convergence to the optimal policy in [10]. The algorithm is presented in pseudo-code in Listing 3.2.

Listing 3.2: one-step Q-Learning

```
initialize Q(s,a) arbitrarily
repeat for each episode:
  initialize s
  repeat for each step of episode:
    choose a from s using policy derived from Q
    take action a, observe r, s'
```

$Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_a Q(s',a) - Q(s,a)]$
s=s'
until s is terminal

Q-Learning can be implemented using a simple tabular approach but this has some flaws with increasing complexity of the model. A function approximation technique can be used with more complex models.

# 4. Used platforms

This chapter describes the platforms and software used in the development of the bots AI. The first section describes Unreal Tournament, the basic rules and the environment. The second section describes Pogamut, which is the platform the bot is implemented on. The last section describes a testing plugin I implemented for Pogamut that I use to measure the bots performance.

## 4.1 Unreal Tournament

Unreal Tournament 2004 (UT) is an arena first-person shooter (FPS) focused on multiplayer deathmatches developed by Epic Games and released in 2004. In UT the player controls a virtual character in a closed environment (map) - see Figure 4.1 - where its main objective is to kill the opponent. There are several other game-modes that do not focus solely on killing the opponent, like Capture The Flag. In this thesis I use only the Deathmatch game-mode.

In UT the player can kill the opponent using a wide variety of weapons. Each in-game character has health, armor, weapons and ammunition for his weapons. Armor acts as extra protection for the character. To damage the health of a character his armor has to be removed first. Both armor and health generally ranges from 0 to 200 and when health is lowered to 0, the character dies. Weapons, health packs, armor packs and ammunition can be picked up on several places on the map. The places that items can be found on are called spawn-points and the items respawn on them after a certain amount of time.

Figure 4.1: Characters in UT ©Epic Games

In Deathmatch (DM), the score of the player is exactly the number of times he killed any opponent. There is a team-based variant of deathmatch - Team-

Deathmatch (TDM) where there are two teams that have their kills pooled into a total score. Capture The Flag (CTF) is a team-based mode where the goal of the team is to pick up the enemy flag in their base and bring it back to their base without dying. In CTF the score is the number of times the team captured the enemy flag. The last major gamemode is Domination in which there are two teams battling for capture points. There is a low amount of capture points and the team gains score for every held capture point over time.

The maps the characters compete in are usually maze-like arenas. Even though some maps feature open space, the characters cannot fly and usually move on the ground. Since the maps are essentially mazes, the maps are represented as navigation graphs within the bot AI. A navigation graph is a graph representing all paths in a map. If a bot moves along edges of a navigation graph it should not collide with any geometry. Navigating the bot then can be solved by several path searching algorithms such as A* [11]. However this system does not provide any information about the geometry of the map and therefore it is impossible for the bot to take cover. Newer games usually use a Navigation mesh [27]. A navigation mesh is a polygon mesh where each polygon represents walkable surface - movement inside a polygon will not cause collisions.

UT has always been mainly a multiplayer game. The game has both a dedicated server and the possibility to host a game on the local computer. It is possible to connect both over a local network and over the Internet. Even though the game is fairly old - released in 2004 - it still has an active player base.

## 4.2 Pogamut

Pogamut is a platform for developing bot AI for UT2004 in Java [14]. Although it is possible to implement bots directly in UT using the UnrealScript scripting language there are many reasons why to use Pogamut for bot AI development. As Pogamut is in Java it is much easier to debug the code and Pogamut provides several helper functions and features to ease development.

Pogamut has five main components: UT2004, GameBots2004, PogamutCore, Pogamut agent and IDE. UT2004 is described in the previous chapter 4.1. GameBots2004 is a modification for UT written in UnrealScript that provides a network text protocol for controlling in-game bots [14]. PogamutCore is a general purpose Java library for connecting agents to almost any virtual environment. The Pogamut agent is a set of classes implementing the necessary classes of PogamutCore.

Pogamut also provides a plugin for NetBeans which makes launching games with the developed AI easier. There are several events that the user can handle in Pogamut by implementing simple event listeners. The helper functions include movement handling, weapon handling and many more.

## 4.3 Testing platform

Since the matches run real-time, I needed to find out a way of running the matches automatically without my supervision, while collecting data from the matches aswell and evolving the policy. Since I wanted to generate performance reports

automatically aswell, I decided to write a Maven site plugin [3] that will run the matches and generate reports.

The plugin uses a Pogamut Addon for running matches directly from Java written by Jakub Gemrot [13]. As the plugin is a Maven site plugin [3], it generates a web-page containing the performance reports of the bots. The metrics provided in the installation are a Frag and Flag count, amount of item, health or weapon pickups and amount of deaths. However it is also possible to write custom metrics in Java. The custom metrics are provided with data that the Pogamut Addon written by Jakub Gemrot outputs.

An example of a report generated by the plugin can be seen in Figure 4.2. In the middle of the figure you can see a graph with the histogram of one of the metrics, graphs for each of the metrics are generated in the report. The report also contains histograms filtered for each of the maps the bot is tested on and then an overall graph for all the matches. In the bottom part of the figure you can see a table with the results of the last experiment with each match on one line. The right-most column contains a link to a replay which is saved for each match from the last experiment.

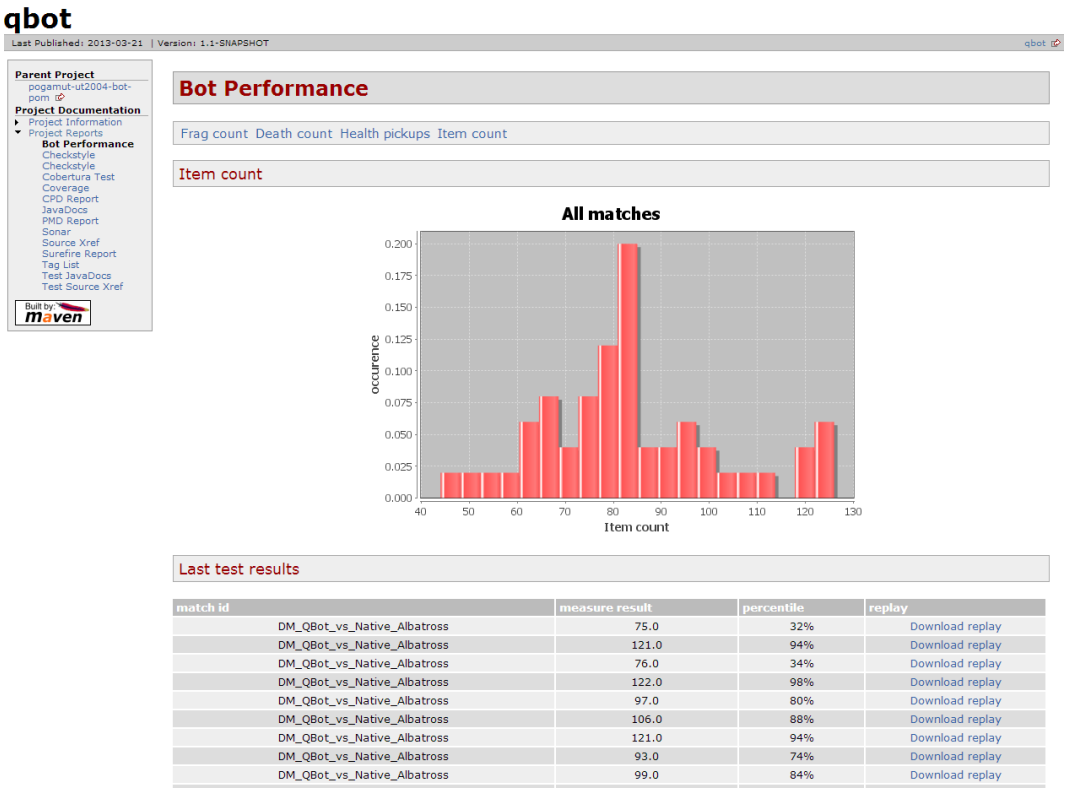



Figure 4.2: An example report generated by the testing plugin

# 5. Design and implementation

This chapter describes the bot architecture and implementation details. The first section describes the design and general implementation architecture. The second section describes how I implemented Q-Learning and the third chapter describes the different selection policies implemented. The last chapter is about the clustering option of the bots state space.

## 5.1 Architecture

This section describes the most important modules in the bots architecture. The inspiration for the bot is a hard-coded bot called HunterBot [2] which is a bot provided for testing with the Pogamut platform. My goal is to evolve a policy using the same state and action space the HunterBot uses and compare the performance. HunterBot has a very simple rule based AI that is presented in Listing 5.1 and the state of HunterBot is described in Table 5.1.

Listing 5.1: Hunter Bot rules in pseudo-code

```
if(enemyVisible and weaponLoaded)
        enemy=visibleEnemy
        combat(enemy)
else if(enemy!=null and seenRecently)
        combat(enemy)
else if(health<90)
        findHealth()
else
        findItems()
```

In total, the state of HunterBot can be saved in seven bits. There are just three actions the HunterBot uses: a combat action, a health action and a item action. The combat action shoots, if there is an enemy visible or looks for the enemy if its not. The health action tries to heal the bot by looking for health packs. The item action just picks a random item and paths to it. Since the health condition is very arbitrary I changed the health state variable to contain health ranges in 2 bits. The ranges are 0-30, 30-80, 80-110 and 110-200 health.

| **state variable description** | **variable size** |
|---|---|
| can the bot see any enemy character | 1 bit |
| is the current weapon loaded with ammunition | 1 bit |
| is the bot shooting his weapon | 1 bit |
| is the bot receiving damage | 1 bit |
| whether the bot has recently seen an enemy | 1 bit |
| whether there is any item visible | 1 bit |
| is the health of the bot lower than 90 | 1 bit |

Table 5.1: HunterBot state

For a simple architecture review, view a simple class diagram in Figure 5.1. I used a BotState container to decouple the QTable from the actual BotState used. The BotState class is a bit array in which you can set fields mapped to the array to certain values and then get the numerical integer value of the bit array to use in the hash-table in the QTable class. The QTable class also uses a policy pattern class to select the action according to several different policies, such as $\epsilon$-greedy, simple exploit etc.

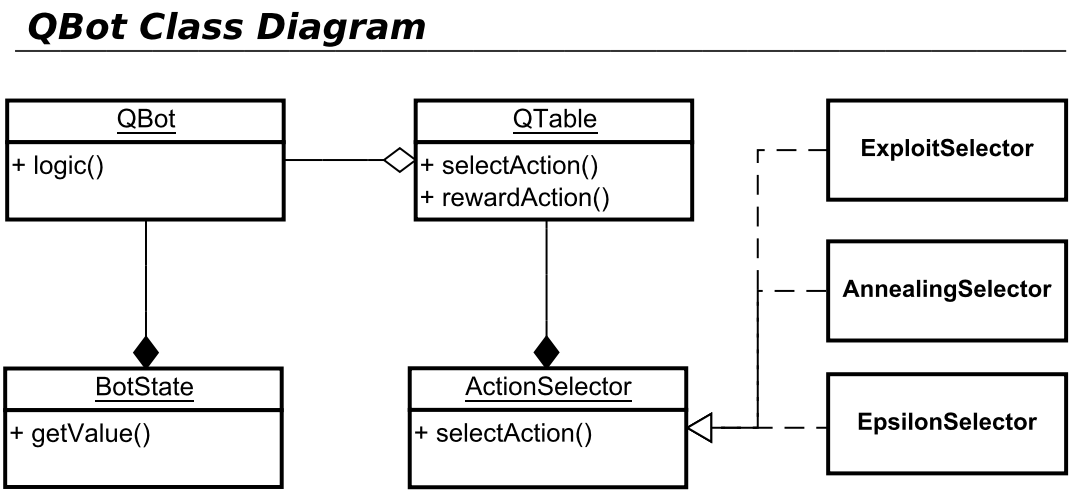



Figure 5.1: Architecture diagram

## 5.2 Q-learning

The learning part of the bot is implemented in the QTable class. To implement the Q-Learning logic i used a tabular approach [10] and save the state-action values in a hash-table in the QTable class. The class uses a hash-table to save the state-action values. The values in the hash-table are lazy-initialized, meaning states that are impossible are not saved and do not take any memory.

There are two main entry points in the class. One is a action-selection function and the second is a reward function. The action-selection function takes a state and returns the action according to the current decision policy. The reward function needs the last state and action taken, the reward and the new state. The function updates the hash-table according to the Formula 3.4.

The reward function i used for the Q-Learning algorithm is a reward function that rewards hitting an enemy and punishes being damaged. In each episode the reward was:

$$R = damageDone * 2 - damageRecieved$$

I implemented several more reward functions but did not manage to test them. The reward functions implement the RewardFunction interface and the reward function used can be easily changed in the QBot class.

## 5.3 Explore/Exploit

The QTable class uses a selection policy provided by a class implementing the ActionSelector interface. ActionSelector is a policy interface with one function - selectAction - that selects a action id dependent on the state-action values as described in Chapter 3.1.

The selectAction signature is as follows: int selectAction(float[] normalized, float max). The normalized array is an array containing the state-action values. The key in the array is the action, the state is not necessary for the decision. The array is normalized by transforming the values to vary from 0 to 1 while conserving their relative ratio. This doesn't affect the decision of the algorithm, actually most selectors would have to normalize the values. I implemented several selectors to test with - ExploitSelector, DistributionSelector, EpsilonSelector and AnnealingSelector.

- **ExploitSelector** - this is the simplest of the selectors I implemented. It always selects the action that has the highest state-action value. This selector has many disadvantages, as it can find a local maximum during the evolution of the function and never reach the optimal function.
- **DistributionSelector** - this selector selects actions according to the relative value of each action-state value. An action $a_i$ with a state-action value $v_i$ has a chance of being selected $p(i) = \frac{v_i}{\sum_1^i v_i}$. This selector has a good amount of exploration involved, but it tends to explore too much and it is not possible to control the rate of exploration very easily. This selector is a Boltzmann selector [17] with a constant explore rate.
- **EpsilonSelector** - is a selection policy well known as $\epsilon$-greedy [26] in which exploration of a random action is selected with a chance of $\epsilon$ and the best valued action is selected with a chance of $1 - \epsilon$. Both advantage and disadvantage of this selection policy is the fact that it explores at a constant rate. With an offline learning algorithm, this is not a disadvantage but with an online learning algorithm it would be better if the rate of exploration would slowly lower with experience.
- **AnnealingSelector** attempts to solve the problem that the $\epsilon$-greedy policy has by combining the ExploitSelector and the DistributionSelector. The DistributionSelector is chosen with a chance that lowers over time and otherwise the action selected is the same as in the ExploitSelector.

## 5.4 Clustering

To reduce the size of the state space, I used a clustering algorithm on the state space and use the clusters generated as the state space instead. Clustering is a way of grouping a certain set of objects together in a group (cluster) so that they are similar to each other. If we apply this to our states we get clusters that we can use as new aggregated states in our Q-Learning algorithm. The clustering algorithm used is a algorithm from the Weka package [20] called SimpleKMeans [19].

The SimpleKMeans algorithm takes $n$ vectors and clusters them into $c$ clusters in which each vector belongs in the cluster that is closest to the vector. The metric we used in this algorithm is simple Euclidean distance. The vector components are normalized by their bounds to ensure all the components have the same weight in the Euclidean distance.

The result of this algorithm is a division of the state space into Voronoi cells [30], each cell represented by the cluster. An example clustering result is presented in Figure 5.2. The plane in the figure is divided into cells, each colored with one color. The squares in the figure represent the vectors used for the clustering algorithm and the circles the centroids of each cluster. For each point in the figure its color corresponds to the closest cluster centroid. If we had two float variables in the state-space of the bot, this could be the result of the clustering algorithm.

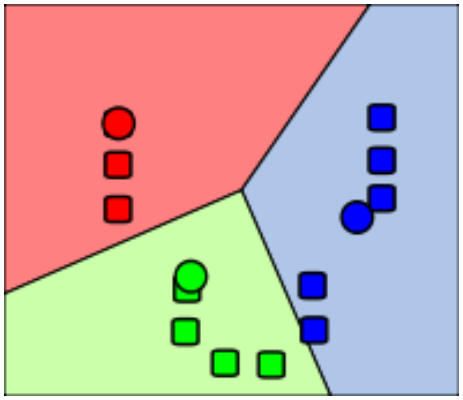


Figure 5.2: Two-dimensional vectors and their clusters represented by a Voronoi diagram — Figure reprinted from : [1]

I built in the clustering into the BotState class to hide the fact that there is clustering going on from the rest of the application. Enabling clustering is therefore very simple and can be enabled in the constructor of BotState. The clustering algorithm is executed during the initialization of the BotState and it needs a log with unclustered states. This means that to run the clustering algorithm, it is necessary to run the bot with unclustered states first. The clustering uses the last one hundred thousand unclustered states logged in the state log and clusters into an amount of clusters specified in the bot configuration file.

What I needed to consider is that the cluster centroids do depend on the behavior of the bot that produced the state set. If a bot is very aggressive the resulting states will probably have a lot more states where the bot has a low health value than if the bot was defensive. Ideally the clustering should be executed on the most recent state set and even each episode during bot execution. However whenever we run the clustering algorithm on a new state set, the resulting state set is different from the one we use in the policy we are evolving. Recreating the Q-Table every episode is not an option because the policy would never evolve. So before each experiment I ran the bot using the most recent iteration to get a new state set and then used that state set for clustering in several matches.

It might be possible to reuse the Q-Table after new clusters are generated. A bijection from the old clusters to the new clusters would have to be created, while still conserving some kind of relevance. This could be solved by applying

the least squares method on the distance between the centroids and minimizing the total distance of the mapping. Even though it could improve the performance I did not explore this possibility in this work.

# 6. Results and discussion

In this chapter I write about the experiment setup, the different iterations of the bot and discuss their performance. The first section describes the way the experiments were setup and ran and the second section presents us with the most simple Q-Learning bot and his performance. The third section 6.3 introduces clustering of the bots state space and measures its impact on performance. Section 6.4 experiments with different learning and discount rates and measures their performance. The last section 6.5 is an overview of the results and discussion.

## 6.1 Experiment setup

To have a good comparison between the different versions of the bot I used the exact same maps and opponents for each bot tested. I used two different maps in the Deathmatch mode - Flux2 and Albatross.

Flux2 is an enclosed map consisting of a number of rooms connected by tunnels, it is moderately large for a 1 vs 1 match. The second map Albatross has a lot more open space as it has more of an arena layout with an open area in the middle of the map and several tunnels around it. Albatross is also smaller than Flux2, but in general it has more difficult features to navigate.

The opponent I used for the experiments was the bot AI provided with Unreal Tournament (NativeBot). These bots have a parameter called skillLevel, which basically controls how fast their reflexes are and how precise they can shoot. To get a predicative performance I needed a skill level that would not cause the matches to be completely one sided for either the NativeBot or HunterBot. I tested against several skill levels and I decided to use skill level three for all matches.

You can view the HunterBot's performance against a skill level three and skill level six in the Figures 6.1 and 6.2. It is clear that matches against a skill level six NativeBot would not give me any good data. The Figure 6.2 presents us with a histogram of the frag count the HunterBot scored during matches against a native bot with a skill level set to six. The problem with this sample is, that we basically have just two results for each match: HunterBot scored at least one frag, HunterBot did not score any frags. Therefore we would not be able to compare any AI that we develop to HunterBot very effectively.

On each map and against each opponent I ran 25 matches and the frag limit for each match was 10 frags. Each of these experiments took about 20 hours of time to run. It is possible to run UT in higher speeds to speed up experiments. However since that usually decreases the performance of any AI developed for a normal game speed I decided not to use a higher game speed.

## 6.2 Basic Q-Learning

First of all i needed to get a baseline performance and since I want to compare my bot to the HunterBot I had HunterBot against a NativeBot. In the Table 6.1 and Figure 6.1 below you can see that he had about half the performance of the

| Frag count statistics - HunterBot | |
|---|---|
| **Sum** | 256 |
| **Mean** | 5.12 |
| **Median** | 5 |
| **Standard deviation** | 2.44 |

Table 6.1: Frag count statistics - HunterBot

Native bot. HunterBot employs rules that are described in Chapter 5. Except that the AI sometimes gets stuck on certain difficult parts of the map the AIs behavior is as described in the rules. I did see the bot I implemented get stuck aswell and is probably caused by defects in the maps navigation graph.

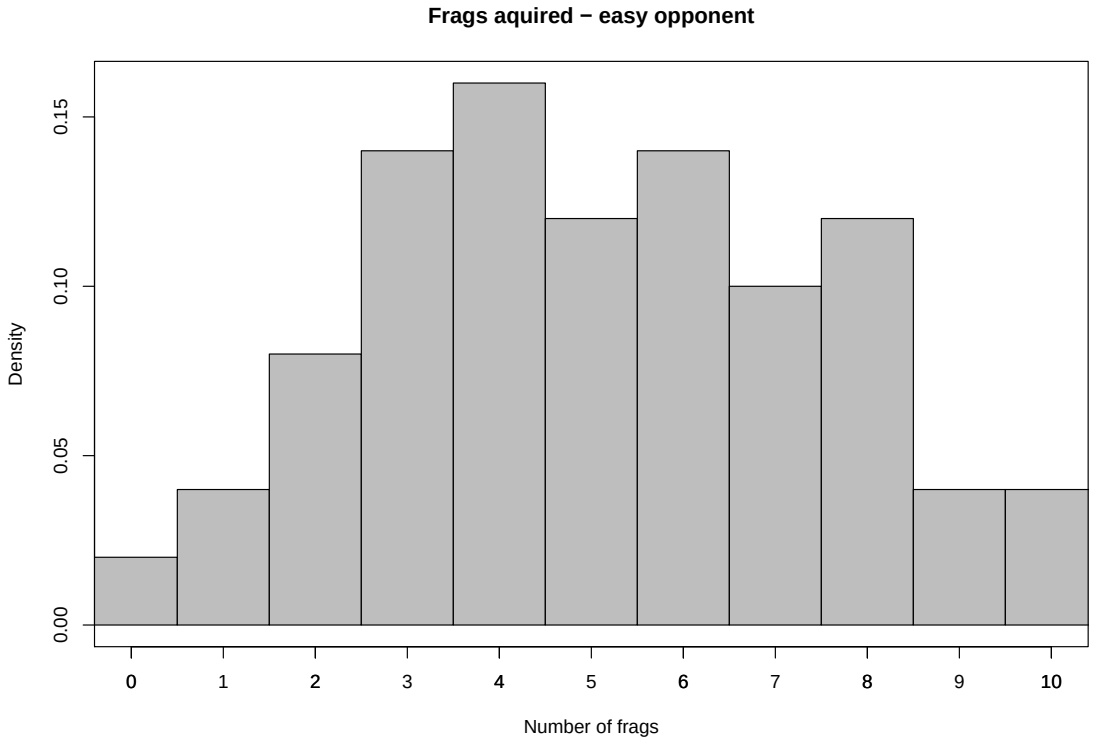



Figure 6.1: Hunter bot against a Native bot (skill level 3)

As discussed in Chapter 6.1 in all the following experiments I use a Native bot with its skill level set to three. The first bot I implemented and tested is a simple Q-Learning bot. The implementation details and its state-space and action-space are described in Chapter 5. I ran the experiment both with a pure exploit selection strategy and a $\epsilon$-greedy strategy. I wanted to see the performance with the most commonly used action-selection policies first before deciding what to do next. The measured performance is displayed in Table 6.2.

This performance is not anything near I hoped it to be. The results in other works implementing a RL FPS bot suggest a performance worse than hard-coded AI, however it is usually at least 1/5 of the performance of a hard-coded bot. There is a slight improvement when using the $\epsilon$-greedy policy in this case, but its very low. The bot did learn to shoot the enemy when he saw an enemy and he

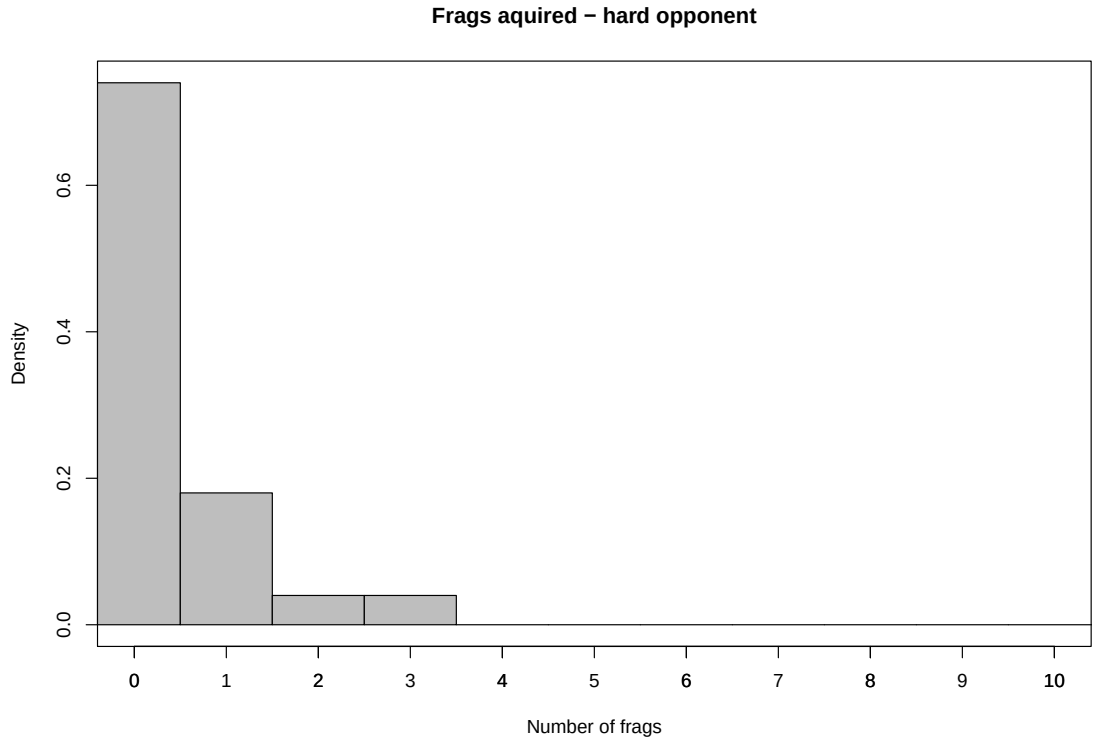



Figure 6.2: Hunter bot against a Native bot (skill level 6)

| **Frag count statistics - Q-Learning** | | | | |
|---|---|---|---|---|
| | **Sum** | **Mean** | **Median** | **Standard deviation** |
| **HunterBot** | 256 | 5.12 | 5 | 2.44 |
| **Exploit QBot** | 5 | 0.1 | 0 | 0.30 |
| **$\epsilon$-greedy QBot** | 8 | 0.16 | 0 | 0.47 |

Table 6.2: Frag count statistics - Q-Learning

did move around the map. However he often switched to an exploratory action from a combat action while being confronted or generally seemed to hesitate a lot. Sometimes the bot even got a little stuck because he was switching from one action to another periodically. It looks that the large size of the state space causes the bot to change his decision too often, and that the preferred action differs too much between small changes to the state.

## 6.3 Clustering

I attempted to solve the problems that the bot from Chapter 6.2 by reducing the state-space size by using clustering. For more details about clustering see Chapter 5.4. The advantages of clustering are that I can decide how much I want to reduce the state space. I tested various amounts of clusters - 5, 10, 20, 50 and 100 reducing the original size of the state-space from 256. You can see the frag count statistics in Table 6.3.

There is a very large increase in performance when using 20 or 10 clusters

| Frag count statistics - Clustering | | | | |
|---|---|---|---|---|
| | **Sum** | **Mean** | **Median** | **Standard deviation** |
| **HunterBot** | 256 | 5.12 | 5 | 2.44 |
| **100 clusters** | 26 | 0.52 | 0 | 0.81 |
| **100 clusters exploit** | 20 | 0.4 | 0 | 0.78 |
| **50 clusters** | 18 | 0.36 | 0 | 0.72 |
| **50 clusters exploit** | 32 | 0.64 | 0 | 0.92 |
| **20 clusters** | 22 | 0.44 | 0 | 0.81 |
| **20 clusters exploit** | 74 | 1.48 | 1 | 1.4 |
| **10 clusters** | 45 | 0.9 | 0 | 1.58 |
| **10 clusters exploit** | 86 | 1.72 | 1 | 1.51 |
| **5 clusters exploit** | 0 | 0 | 0 | 0 |

Table 6.3: Frag count statistics - Clustering

with an exploit selection policy. The bot with 10 and 20 clusters behaved a lot more coherent, he had clear decisions and did not hesitate as the last bot did. The main problem with his performance is in my opinion the fact that he does not change his decisions according on how powerful weapons he currently has. Many times he died to the opponent because he fought with the least powerful gun instead of finding a more powerful gun instead. With clustering set to 5 clusters, the bot did not kill the enemy even once. Five clusters seems to be below the treshold where the bot can learn anything useful.

The most important cluster centroids created by the SimpleKMeans clustering algorithm are presented in Tables 6.4 and 6.5. The Share row denotes how much of the state set the cluster covers and the share is how I selected the clusters to include in the tables. The health variable ranges from 0 to 200, however its impact on the distance used in the clustering is the same as the impact of the other variables as described in Chapter 5.4. The clustering algorithm normalizes the variables using the range they span.

In both cases the cluster centroids are similar. The most common clusters are clusters of states where there is no combat possible, states with the Enemy in sight variable set to 0 and rest of the variables set to nominal values except for item in sight which results in an extra cluster. Combat states are very uncommon (less than 10%) and are usually tied with a lower health value - all centroids with an Enemy in sight have a health lower than 75.

The fact that the bot can perform fairly well with this little distinction between the cluster centroids suggests that the main problem with the previous bots was in fact slow learning. That is because reducing the complexity of the state/action-space improves the learning rate and in my case performance. However the exploring action selection policy does not perform higher than a simple exploit selection policy which is unexpected. The small size of the state-space causes the policy to evolve very fast with a exploit selection policy and the exploring in the long run just lowers the performance.

Therefore I made a selection policy called annealing that will explore in the beginning of the game and switch to a pure exploit selection after that. The performance measured is in the Table 6.6 below. However even this selection policy does not outperform the exploit policy. The selection I used explores for

| Clustering Centroids - 10 clusters | | | | | |
|---|---|---|---|---|---|
| **Share** | 0.38 | 0.27 | 0.13 | 0.04 | 0.03 |
| **Enemy in sight** | 0 | 0 | 0 | 1 | 1 |
| **Weapon loaded** | 1 | 1 | 1 | 1 | 1 |
| **Shooting** | 0.002 | 0.002 | 0 | 0.3 | 0 |
| **Being damaged** | 0.0112 | 0.01 | 0 | 0.1 | 0.1 |
| **Last enemy visible** | 0 | 0 | 0 | 0.6 | 0 |
| **Item in sight** | 1 | 0 | 1 | 0.4 | 1 |
| **Health** | 98.6 | 99.3 | 114.0 | 70.9 | 75.6 |

Table 6.4: Clustering Centroids - 10 clusters

| Clustering Centroids - 20 clusters | | | | | |
|---|---|---|---|---|---|
| **Share** | 0.28 | 0.27 | 0.03 | 0.02 | 0.02 |
| **Enemy in sight** | 0 | 0 | 1 | 1 | 1 |
| **Weapon loaded** | 1 | 1 | 1 | 1 | 1 |
| **Shooting** | 0 | 0 | 0 | 0.77 | 0 |
| **Being damaged** | 0 | 0 | 0.1 | 0.1 | 0.1 |
| **Last enemy visible** | 0 | 0 | 0 | 1 | 0 |
| **Item in sight** | 1 | 0 | 1 | 0.6 | 0 |
| **Health** | 99.8 | 99.5 | 75.6 | 70.7 | 72.8 |

Table 6.5: Clustering Centroids - 20 clusters

about 1/5 of the game before switching to a pure exploit policy. For a closer look at the performance of the bot with 10 and 20 clusters, see the Figures 6.3 and 6.4.

## 6.4 Learning and discount rate tuning

The next iteration is just about fine tuning the learning and discount rate parameters of the Q-Learning algorithm. The learning rate ($\alpha$) determines how will the reward override the current value of the policy function. A learning rate of 1 would make the policy use only the latest reward as its state-action value and a learning rate of 0 would never change the state-action value. The discount

| Frag count statistics - Annealing Selector | | | | |
|---|---|---|---|---|
| | **Sum** | **Mean** | **Median** | **Standard deviation** |
| **HunterBot** | 256 | 5.12 | 5 | 2.44 |
| **20 clusters** | 22 | 0.44 | 0 | 0.81 |
| **20 clusters exploit** | 74 | 1.48 | 1 | 1.4 |
| **20 clusters annealing** | 41 | 0.82 | 0 | 1.1 |
| **10 clusters** | 45 | 0.9 | 0 | 1.58 |
| **10 clusters exploit** | 86 | 1.72 | 1 | 1.51 |
| **10 clusters annealing** | 32 | 0.64 | 0 | 0.88 |

Table 6.6: Frag count statistics - Annealing Selector

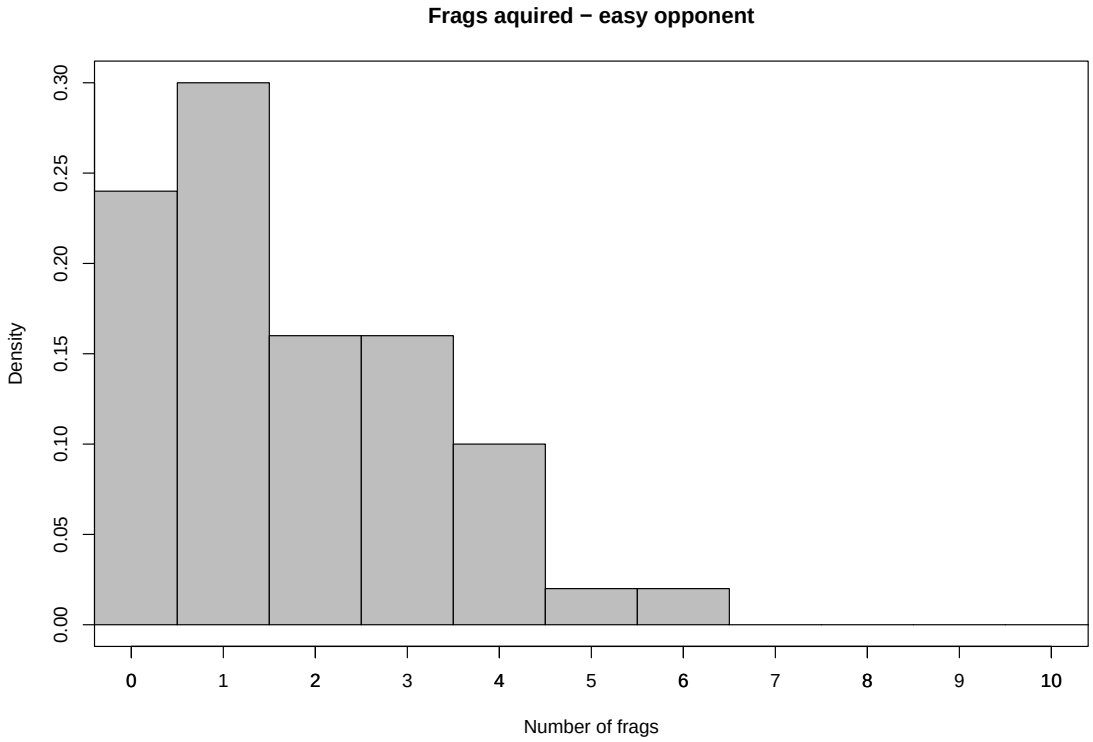



Figure 6.3: Exploit QBot with clustering (10 clusters) against a Native bot (skill level 3)

| **Frag count statistics - Learning/Discount rate** | | | | |
|---|---|---|---|---|
| | **Sum** | **Mean** | **Median** | **Standard deviation** |
| $\alpha$=**0.9** $\gamma$=**0.5** | 74 | 1.48 | 1 | 1.4 |
| $\alpha$=**0.9** $\gamma$=**0.7** | 75 | 1.5 | 1 | 1.6 |
| $\alpha$=**0.9** $\gamma$=**0.2** | 68 | 1.36 | 1 | 1.5 |
| $\alpha$=**0.8** $\gamma$=**0.4** | 61 | 1.22 | 1 | 1.2 |
| $\alpha$=**0.7** $\gamma$=**0.3** | 65 | 1.3 | 1 | 1.5 |

Table 6.7: Frag count statistics - Learning/Discount rate

rate($\gamma$) controls the importance of planning in the bots behavior. A discount rate of 1 would cause the bot to plan ahead and a value of 0 would make the bot opportunistic and never propagate the reward to past states.

All experiments were ran with clustering set to 20 clusters and an exploit selection strategy. The results in Table 6.7 all the learning rates tested perform fairly similar. The results also show us that the performance does not depend on planning very much with a discount rate of 0.2 performing as well as a discount rate of 0.7.

The bot behaves very similar regardless on what values of learning/discount rate he uses which might be because in FPS games, the best performance depends more on reflexes than planning. He shoots the enemy on sight regardless of what weapon he has available and if there is no enemy visible he explores the map.

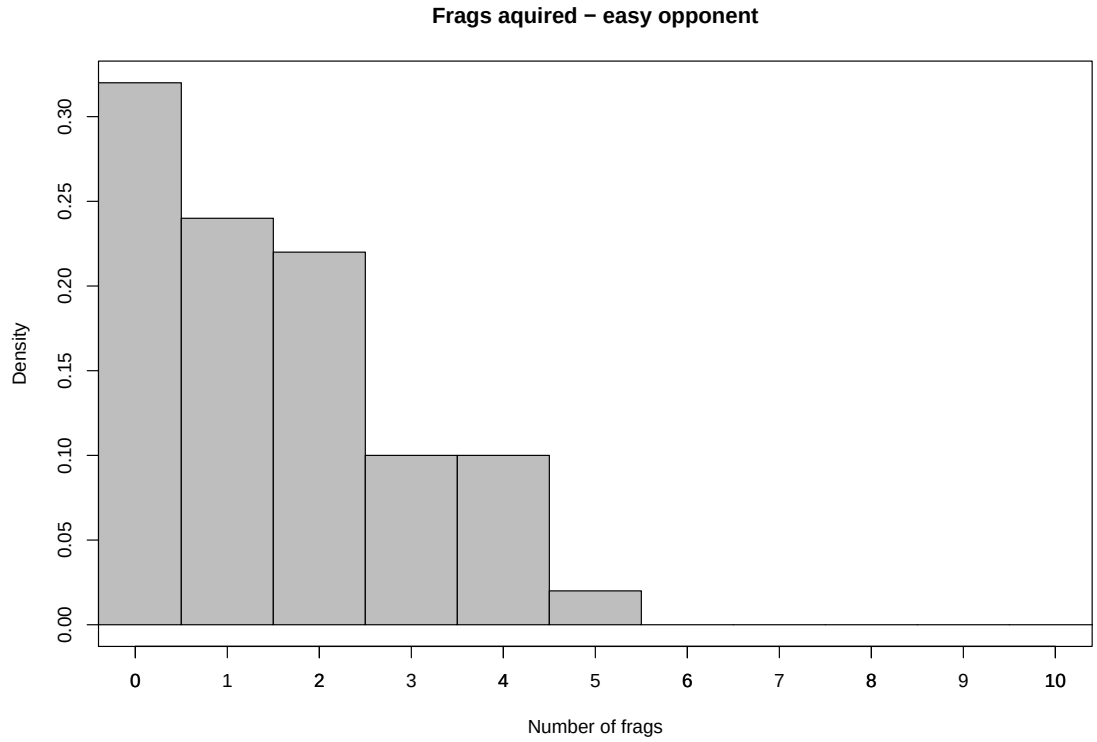



Figure 6.4: Exploit QBot with clustering (20 clusters) against a Native bot (skill level 3)

## 6.5 Discussion

The behavior the Q-Learning algorithm developed could be described very easily in a few simple rules. If an enemy is in sight, shoot it, if it is not, either go look for health packs or items. Since this is very similar to the rules that the HunterBot uses, it seems that its not possible to develop better performing rules with the sensory information provided to the HunterBot. The difference however is in the small things. Even with the general behavior of the Q-Learning bot matching the HunterBot, its performance is lower at about 1/3 of the HunterBot performance.

The difference in performance is caused by many small defects in the bots behavior, especially changing actions erratically. Another big defect that is visible in the bots behavior is that he does not change his behavior depending on the weapons he picked up. This variable is not included in the state space of HunterBot, so it is not included in the QBot's state space and the behavior cannot change according to weapons available. But, HunterBot usually does not have this problem so its possible QBot does not explore the level as effectively as HunterBot.

Nevertheless a performance the QBot is not uncommon in other similar implementations of RL. In [22] Michelle McPartland and Marcus Gallagher develop three different RL bots. A simple FlatRL bot, a HierarchicalRL bot and a RuleBasedRL bot. The FlatRL bot performs at 1.9 average frags on simple maps and 1.1 on more complicated maps, while the hard-coded bot they compare it to performs at an average of 14.1 frags per match on simple maps and 9.8 on complicated maps. The other approaches they use perform better, with the Hi-

erarchicalRL bot performance being 5 frags (3.9 on complicated) on average per match and the RuleBasedRL bot achieving 7.4 frags (4.4 on complicated) per match. Our bots performance is therefore about the same as the performance of the HierarchicalRL bot.

In [8] Remco Bonse, Ward Kockelkorn, Ruben Smelik, Pim Veelders and Wilco Moerman implemented a Q-Learning subsystem for a hard-coded bot that controls his combat movement. Their experiments showed that by learning combat movement they increased the performance of their bot by about 15%. Their AI however did not learn online and had to learn for a very long period of time to start performing well.

In [15] Ahmed S. Hefny, Ayat A. Hatem, Mahmoud M. Shalaby and Amir F. Atiya develop a hierarchical model for the CTF gamemode, where higher level actions are selected by Q-Learning and lower level actions are selected by neural nets. The performance results presented are very unclear, but the authors claim that reinforcement learning can be used to develop team behavior that adapts to the enemy team strategy. A similar, but much more clear result is presented in [21] where Megan Smith, Stephen Lee-Urban and Héctor Muñoz-Avila create an online RL algorithm that teaches a team of bots a team strategy for the Dominate game-mode. The results presented in this work are very good, competitive even against difficult hard-coded team strategies. However the state space of this RL problem is extremely small (3 bits).

The results from other related works suggest that Q-Learning excels at improving tasks simple in complexity, best working when learning a small subset of the complete problem. In most cases, good performance was achieved by learning a low level subsystem of the bots behavior. In my work, I attempted to learn just the high-level decision process to see if the concept can be extended to them as presented in the strategy based policies in CTF and Domination modes. However even though my bot did learn meaningful rules, the AI overall did not perform very well. Even so the performance was comparable to other similar implementations.

# 7. Future work

This section describes additional work that could be done to improve the performance of the Q-learning agent developed in this thesis.

## 7.1 Eligibility traces

Eligibility traces is a very useful mechanism of reinforcement learning which could be used to improve the performance of the AI I implemented. Eligibility traces is a way to propagate the reward function value to a longer chain of state-action pairs. The method enables the reward to modify the policy function of the whole path the AI has taken so far at the time of receiving the reward. The behavior I developed does suggest that it might not be necessary to use any planning for good performance. Due to the nature of the Deathmatch mode a reactive behavior is usually preferred over planning. Nevertheless this is just an assumption and work with eligibility traces would be justified.

In the eligibility trace versions of temporal difference learning algorithms, there is additional memory associated with each state, its eligibility trace. The eligibility trace of any state visited is incremented by 1 and each step the eligibility traces of all states decay by $\gamma\lambda$ where $\lambda$ is called the trace-decay parameter. Let $e_t(s)$ be the eligibility trace of state $s$ in time $t$. The eligibility trace value represents the eligibility of a state for applying the learning changes from the reward received. When a reward even occurs we take the TD error

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{7.1}$$

and use it to update all state values according to the following formula

$$V(s) = V(s) + \alpha\delta_t e_t(s) \tag{7.2}$$

This concept can be extended to all TD algorithms such as Sarsa and Q. There are several variants of Q-Learning with eligibility traces [28], [23], [26] which could be used instead of the one-step Q-Learning algorithm that I used in my agent. The possible improvements could be faster learning and better planning.

## 7.2 Reward function

Another area of possible improvement is the reward function. As the reward function is what RL uses to evolve the policy it very much affects the performance of the bot. In my AI I used the same reward function for all bots and did not explore the possibility for any other reward function. The reward function I used in my implementation is described in Chapter 5.2

However as the reward function is very problem specific, there are no guidelines to use while designing the reward function which results in much guesswork. Nevertheless creating several different reward functions and comparing their impact on the performance is a possible improvement to the bots performance.

## 7.3 State space

The state-space and the action-space in my implementation are copying the state and action space of the HunterBot. I made this decision to get a good comparison between a rule based AI and RL. Also, I wanted to see if I can evolve a policy that would work better than the rules controlling the behavior of Hunter and could be potentially translated into rules. Therefore the state-action space are not designed with RL in mind and since the state-space was too large I had to lower the size with clustering. Even though this increased the overall performance, it could be considered a workaround and this problem could be instead solved by designing a better state-space.

One of the possible ways to lower the size of the state-space without clustering would be to create a hierarchical model. A hierarchical model would be a model where a smaller state-space controls the higher level action decision process and for each action there is a different state-space and different policy evolved. However this would in turn cause the bot developed to be completely different from HunterBot, even though this approach showed good performance in [21].

# Conclusion

This work has designed, implemented and tested an adaptive bot behavior. As such the main objective of this thesis has been fulfilled. The final performance of the bot is about one third of the performance of a rule based AI. The performance is comparable with other similar implementations [21,22]. The work experimented with several different action selection policies and the best performing policy in the experiments was a pure exploit policy. During the whole development to run experiments I used a testing framework I implemented for the Pogamut platform.

In the future the bot could be improved by implementing an eligibility-trace version of Q. Also it is possible to extend this approach to a more complex behavior by introducing state variables relevant for more complex behavior. By clustering the states we showed that many variables of the original state were not necessary for a better performing behavior.

# Bibliography


[1] Clusters in a voronoi diagram. http://en.wikipedia.org/wiki/File:K_Means_Example_Step_4.svg.

[2] Hunterbot. svn://artemis.ms.mff.cuni.cz/pogamut/trunk/project/Archetypes/UT2004/04-HunterBot.

[3] Maven site plugin. http://maven.apache.org/plugins/maven-site-plugin/.

[4] Robocup. http://www.robocup.org/.

[5] Christophe Andrieu. An introduction to mcmc for machine learning. 2003.

[6] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artif. Intell.*, 72(1-2):81–138, jan 1995.

[7] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artif. Intell.*, 134(1-2):201–240, jan 2002.

[8] Remco Bonse, Ward Kockelkorn, Ruben Smelik, Pim Veelders, and Wilco Moerman. Learning agents in quake iii.

[9] Alex J. Champandard. A-life, emergent AI and S.T.A.L.K.E.R. http://aigamedev.com/open/interviews/stalker-alife/.

[10] Peter Dayan Chris Watkins. Q-learning. *Machine Learning*, pages 279–292, 1992.

[11] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of a*. *J. ACM*, 32(3):505–536, July 1985.

[12] David A. Ferrucci, Eric W. Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John M. Prager, Nico Schlaefer, and Christopher A. Welty. Building watson: An overview of the deepqa project. *AI Magazine*, 31(3):59–79, 2010.

[13] Jakub Gemrot. Ut2004tournament. svn://artemis.ms.mff.cuni.cz/pogamut/trunk/project/Addons/UT2004Tournament.

[14] Jakub Gemrot, Rudolf Kadlec, Michal Bída, Ondřej Burkert, Radek Píbil, Jan Havlíček, Lukáš Zemčák, Juraj Šimlovič, Radim Vansa, Michal Štolba, Tomáš Plch, and Cyril Brom. Pogamut 3 can assist developers in building ai (not only) for their videogame agents. In Frank Dignum, Jeff Bradshaw, Barry Silverman, and Willem Doesburg, editors, *Agents for Games and Simulations*, volume 5920 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2009.

[15] Ahmed S. Hefny, Ayat A. Hatem, Mahmoud M. Shalaby, Amir F. Atiya, and Amir F. Atiya. Cerberus: Applying supervised and reinforcement learning techniques to capture the flag games. *AIIDE*, 2008.

[16] Feng-Hsiung Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA, 2002.

[17] Mark Humphrys. Action selection methods using reinforcement learning, 1996.

[18] Gal A. Kaminka, Manuela M. Veloso, Steve Schaffer, Chris Sollitto, Rogelio Adobbati, Andrew N. Marshall, Andrew Scholer, and Sheila Tejada. Gamebots: a flexible test bed for multiagent team research. *Commun. ACM*, 45(1):43–45, January 2002.

[19] T. Kanungo, Mda c, D. M. Mount, C. Piatko, N. S. Netanyahu, A. Y. Wu, and R. Silverman. The analysis of a simple k-means clustering algorithm. 2000.

[20] Geoffrey Holmes Bernhard Pfahringer Peter Reutemann Ian H. Witten Mark Hall, Eibe Frank. The weka data mining software: An update. *SIGKDD Explorations*, 11, 2009.

[21] Héctor Munoz-Avila Megan Smith, Stephen Lee-Urban. Retaliate: Learning winning policies in first-person shooter games. *IAAI*, pages 1801–1806, 2007.

[22] Marcus Gallagher Michelle McPartland. Reinforcement learning in first person shooter games. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(1):43–56, 2011.

[23] Jing Peng and Ronald J. Williams. Incremental multi-step q-learning. In *Machine Learning*, pages 226–232. Morgan Kaufmann, 1996.

[24] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[25] Budhitama Subagdja, Wenwen Wang, Ah-Hwee Tan, Yuan-Sin Tan, and Loo-Nin Teow. Memory formation, consolidation, and forgetting in learning agents. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '12, pages 1007–1014, Richland, SC, 2012. International Foundation for Autonomous Agents and Multiagent Systems.

[26] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2005.

[27] Wouter Van Toll, Atlas F Cook, and Roland Geraerts. Navigation meshes for realistic multi-layered environments. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3526–3532. IEEE, 2011.

[28] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[29] James Wexler. Artificial Intelligence in Games: A look at the smarts behind Lionhead Studio's "Black and White" and where it can and will go in the future. 2002.

[30] Wikipedia. Voronoi diagram. http://en.wikipedia.org/wiki/Voronoi_diagram.

[31] Tomas Witzany. Maven testing plugin for pogamut. http://diana.ms.mff.cuni.cz:8080/view/Addons/job/UT2004MavenTestPlugin%20%28deploy%20site%29/site/.

[32] Sule Yildirim and Sindre Berg Stene. A survey on the need and use of ai in game agents. SpringSim '08, pages 124–131. Society for Computer Simulation International, 2008.

# List of Tables



# List of Figures



# List of Algorithms

# Attachments

# CD-ROM

The enclosed CD contains source codes of the QBot and a PDF version of this text. Structure of the CD and instructions on how to replicate experiments are described in the readme file in the root directory.