12-2009

Improving Computer Game Bots' behavior using Q-Learning

Purvag Patel
Southern Illinois University Carbondale, purvag2003@gmail.com

Improving Computer Game Bots' behavior using Q-Learning

by

Purvag Patel

B.E. Computer Engineering

Gujarat University

5th November 2009

A Thesis

Submitted in Partial Fulfillment of the Requirements for the

Master's of Science Degree in Computer Science

Graduate School

Southern Illinois University Carbondale

December 2009

THESIS APPROVAL

Improving Computer Game Bots' behavior using Q-Learning

by

Purvag Patel

B.E. Computer Engineering

Gujarat University

A Thesis

Submitted in Partial Fulfillment of the Requirements for the

Master's of Science Degree in Computer Science

Approved By:

Dr. Norman Carver, Committee Chair

Dr. Michael Wainer

Dr. Henry Hexmoor

Graduate School

Southern Illinois University Carbondale

5th November 2009

AN ABSTRACT OF THE THESIS OF

Purvag Patel, for the Masters of Science degree in Computer Science, presented on November 05, 2009, at Southern Illinois University Carbondale.

Title: Improving Behaviour of Computer Game BOTs' Using Q-Learning

MAJOR PROFESSOR : Dr. Norman Carver

In modern computer video games, the quality of artificial characters plays a prominent role in the success of the game in the market. The aim of intelligent techniques, termed ***game AI***, used in these games is to provide an interesting and challenging game play to a game player. Being highly sophisticated, these games present game developers with similar kind of requirements and challenges as faced by academic AI community. The game companies claim to use sophisticated game AI to model artificial characters such as computer game ***bots***, intelligent realistic AI agents. However, these bots work via simple routines pre-programmed to suit the game map, game rules, game type, and other parameters unique to each game. Mostly, illusive intelligent behaviors are programmed using simple conditional statements and are hard-coded in the bots' logic. Moreover, a game programmer has to spend considerable time configuring crisp inputs for these conditional statements. Therefore, we realize a need for machine learning techniques to dynamically improve bots' behavior and save precious computer programmers' man-hours. So, we selected ***Q-learning***, a reinforcement learning technique, to evolve dynamic intelligent bots, as it is a simple, efficient, and online learning algorithm. Machine learning techniques such as reinforcement learning are know to be intractable if they use a detailed model of the world, and also requires tuning of various parameters to give satisfactory performance. Therefore, for this research we opt to examine Q-learning for evolving a few basic behaviors viz. learning to fight, and planting the bomb for computer game bots. Furthermore, we experimented on how bots would use knowledge learned from abstract models to evolve its behavior in more detailed model of the world. Bots evolved using these techniques would become more pragmatic, believable and capable of showing human-like behavior. This will provide more realistic feel to the game and provide game programmers with an efficient learning technique for programming these bots.

ACKNOWLEDGEMENT

I would like to thank Dr. Norman Carver for being my advisor and providing me with valuable advises and guidance along with his tireless efforts which paved way to my thesis. I would like to thank Dr. Henry Hexmoor for being part of my committee and providing valuable suggestions. Additionally, I appreciate Dr. Michael Wainer's valuable inputs and suggestions. I would also like to thank my family for their valuable support and inspiration they provided me to work harder and achieve success with my endeavors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

It is incontrovertible that the computer and video game industry has become a billion dollar industry which is still growing by leaps and bounds. With the development of 3D game engines, gaming technologies have reached new heights of success. With the growing power of CPUs, and computers becoming cheaper, it's time for game developers to dedicate significant CPU cycles to employ academic AI techniques. Moreover, the quality of artificial characters in a game plays a prominent role in its market value. Games are becoming more and more sophisticated in the simulation of artificial worlds to cater to contemporary game players, who want more believable, realistic, and intelligent agents for games.

Most of the games utilize some form of techniques, called ***game AI***, to simulate intelligence. Perhaps the most prominent *game AI* technique to simulate intelligence is cheating. For example, with the increase in level, the developer will increase the hit-points of non-human player. A game developer is primarily interested in only the engineering side, building algorithms that make game characters appear more realistic or human-like. The aim of agents in *game AI* is like a machine used in the Turing test[1], which humans cannot identify whom they are answering to. So in a broader sense, anything that gives intelligence to an appropriate level to make it more challenging, interesting, and immersive, can be considered as *game AI*. Hence, the goal of agents in *game AI* is to get defeated by its human counterpart in more impressive, unprecedented, and unpredictable manner.

Again, modern games are highly sophisticated and present a wide variety of challenges where academic AI techniques can be used. Common features of such games are[14]:

- real time : games are highly bounded by the response time. They have to compete with the response time of human players and many techniques developed in academic AI are known to be computationally expensive.
- incomplete knowledge : agents in games don't have complete knowledge about the world, for example, current location of the opponents as well as of their own team players.
- limited resources : resources for the agent are limited, for example, the number of bullets in the gun.

[1]A test proposed by British mathematician Alan Turing, and often taken as a test of whether a computer has humanlike intelligence. If a panel of human beings conversing with an unknown entity (via keyboard, for example) believes that that entity is human, and if the entity is actually a computer, then the computer is said to have passed the Turing test[24].

- planning : normally agents in games are goal based, having a predefined sets of goals.

- learning : agent can learn through experience from the game play.

Here, we can see that similar requirements and constraints are faced by researchers in academic AI. Many techniques of academic AI are waiting to be studied and implemented by game developers. Furthermore, games provide a ready testbed to test various algorithms of academic AI. Games are ready with simulation of actual world environments equipped with real world objects such as boxes, walls, gates, etc.. A wide variety of entities can be simulated in such environments, for example, one can test effectiveness of military robots in the battle field with various obstacles.

Game companies claim to be using sophisticated techniques in current games, but they only imitate intelligence using classical techniques such as conditional statements. This can result in a game in which a player may lose interest in due course, as with ***bot***, an artificial Non-player character(NPC), used in various First-person shooter(FPS) games like Counter-Strike[17], Half-Life[18] and Quake[20]. These bots are usually modeled using Finite-State machines(FSM) and programmed using simple if-else statements, resulting in a very predictable bot to an experienced game player [15].These bots play fixed strategies, rather than improving as a result of the game play. Moreover, designing such bots is time consuming because game developers have to configure many crisp values in their hard-coded if-else statements.

We believe that methods of machine learning, extensively studied by academic AI community, could be used in games to address the limitations of current approaches to building bots. The advantages of using a machine learning technique to improve computer games bots' behaviors are:

- it would eliminate/reduce the efforts of game developers in configuring each crisp parameter and hence save costly man-hours in game development, and

- it would allow bots to dynamically evolve their game play as a result of interacting with human players, making games more interesting and unpredictable to human game players.

For this research, we chose to study the use of Q-learning, a type of reinforcement learning technique(RL), to improve the behavior of game bots. We chose Q-learning because it is relatively simple and efficient algorithm, and it can be applied to dynamic online learning. Our original plan was to try to experiment with source code of the FPS game Counter-Strike, but we found this impractical. Instead, we developed our own game platform for experimentation, a highly simplified simulation of FPS games like Counter-Strike.

While machine learning techniques can be easy to apply, they can become intractable if they use detailed models of the world but simplified, abstract models may not result in acceptable learned performance. Furthermore, like most machine learning techniques, RL has a number of parameters that can greatly affect how well the technique works, but there is only limited guidance available for setting these parameters. For this thesis, we set out to answer some basic questions about how well reinforcement learning might be able to work for FPS game bots. We focused on the following three sets of experiments:

1. learning to fight: seeing if and how well bots could use RL to learn to fight, and how the resulting performance would compare to human programmed opponent bots,
2. learning to plant the bomb: instead of rewarding bots for fighting, what would happen to bots' behavior if they were rewarded for accomplishing the goal of planting bombs, and
3. learning for deployment: if bots initially learn using abstract states models (as might be done by the game designers), how does initializing their knowledge from the abstract models help in learning with more detailed models.

The ultimate goal of these experiments is to evolve sophisticated and unpredictable bots which can be used by game developers and provide unprecedented fun to game players.

2 BACKGROUND

During the first phase of research we surveyed the nature of game AI used in video games along with the level of academic AI technique used in few selected games.

2.1 Game AI

All games incorporate some form of game AI in them. Since the start of game development, game developers have always used game AI for game characters to make them appear intelligent. It can be ghosts in the classic game of PAC man or sophisticated bots in first-person shooter game Half-life [1].

To date, the most commonly used game AI technique is cheating. Partly, the reason is that the goal of the game AI is quite different than AI studied in academics. In academics we aim to create AI entities that reach human-level intelligence to outperform human beings. But this high-level of intelligence is not acceptable in game AI because such high-level of intelligence might not give human players a chance to win. So, if a player cannot advance in levels or if game play does not progress eventually a player will lose interest in the game. So, one can formulate that the goal of game AI is to be knocked out by human player in most challenging, impressive and unprecedented manner.

Human players while playing against or with computer players, which are used to replace humans, have a few expectations. Bob Scott lists out these expectations which may act as guidelines for game developers [2]:

1. *Predictability and unpredictability:* Human actions are unpredictable, and at the same time they can be very predictable. These kinds of actions tend to provide surprise elements to the game. For example, in FPS this means that throwing granite for the purpose of distraction is unpredictable. An example of predictable behavior is using the same route repeatedly. Also, human actions are very inconsistent and sometimes they tend to behave stupidly. Mimicking this form of behavior in a game can be difficult.

2. *Support:* Often, a computer player has to provide support to fellow human players. This can be as simple as guiding human players through the game or providing cover during combat. An efficient method of communication is necessary between computers and human players which is hard to achieve.

3. *Surprise:* The surprise element is most talked about among game players. Surprise can be in many forms, viz. harassment, ambushes and team support.

4. *Winning, Losing and Losing Well:* As we said earlier that it is relatively easy to engineer a computer player that always wins or loses against a human player. The main focus herein is believability of the opponents. Difficulty settings provided in almost every game allow game players to set the level of computer player whether it will be challenging or not. Also, there are few games wherein the computer players change its difficulty levels based on number of wins or lose.

5. *Cheating:* Whether a game AI should be allowed to cheat or not has always been debatable. Objections are that AI will have unfair advantage over humans, but it's also true that computers are already at a disadvantage because they are impassive. To our point of view, cheating is acceptable as long as it does not get detected by a human player keeping computer player interesting and challenging.

Hereby, we showed the level and the form of artificial intelligence that is desired in game AI. This is not to say that academic AI techniques have been completely absent from game AI.

2.2 Current Usage of AI in Games

We wanted to study the level of AI currently used in video games. So, we selected a few of the most popular games and examined them. It turns out that the majority of AI is scripted or hardcoded in logic without using any academic AI techniques.

2.2.1 Age of Empires

Age of Empires is a real-time strategy game. In Age of Empires a player builds kingdoms and armies with a goal to survive through ages, defend his own kingdom and conquer other kingdoms. A player starts with few workers and his/her primary building and then slowly builds a complete town and many buildings.

AI of Age of Empires is smart enough to find and revisit other kingdoms and their buildings, i.e. pathfinding, and also start attacking them. One of the major drawbacks with this game is that AI is too predictable. A player can easily predict the path of AI which tends to repeat the same path repeatedly giving human players an edge. Also, AI is very poor when it comes to self-defense.

2.2.2 Half-life

Half-life is a science fiction first-person shooter game (FPS). Half-life takes shooter games to the next level of intelligent behavior of enemy monsters and won 35 ‘game of the year’ awards in 2004. Half-life is a more or less a traditional game in which a player advances by killing creatures coming in his way. Monsters in this game can hear, see, and track human players. Along with this they can also flee when they are getting defeated and call for help from other monsters to trap a human player.

2.2.3 Creature

Creature is an artificial life program. Software comes with six eggs, each with unique creatures in them. The game starts when an egg hatches and norms come out. Norms learn by interacting with the environment and a player watches them grow through interactions. A norm passes through adolescence and adulthood and then eventually lays eggs. A player builds colonies of such creatures.

Being first in its kind, this is the first real use of artificial intelligence techniques in video games. It uses machine learning technique, namely neural network, as the learning algorithm for norms.

2.2.4 Black and White

Black and White, a god game, is one of the most widely discussed among academic AI researchers working on game AI. A player plays a god in this game and the world changes according to actions of a player. The game starts with a baby creature which is a player’s pet. The creature learns from the feedback and actions of a player. For example, a player can teach the creature that eating fellow humans is a bad by slapping it, and can make them learn that helping the same people is good by pampering it when it helps them. The creature in the game is very realistic and performs various bodily functions such as sleeping, eating, playing, etc. At the end of game creature becomes evil or good based on the feedback it had received from the player.

Black and White makes extensive use of a decision tree for learning which is based on Quinlan’s ID3 system. This game has received a Guinness world record for creature technology, along with few other awards.

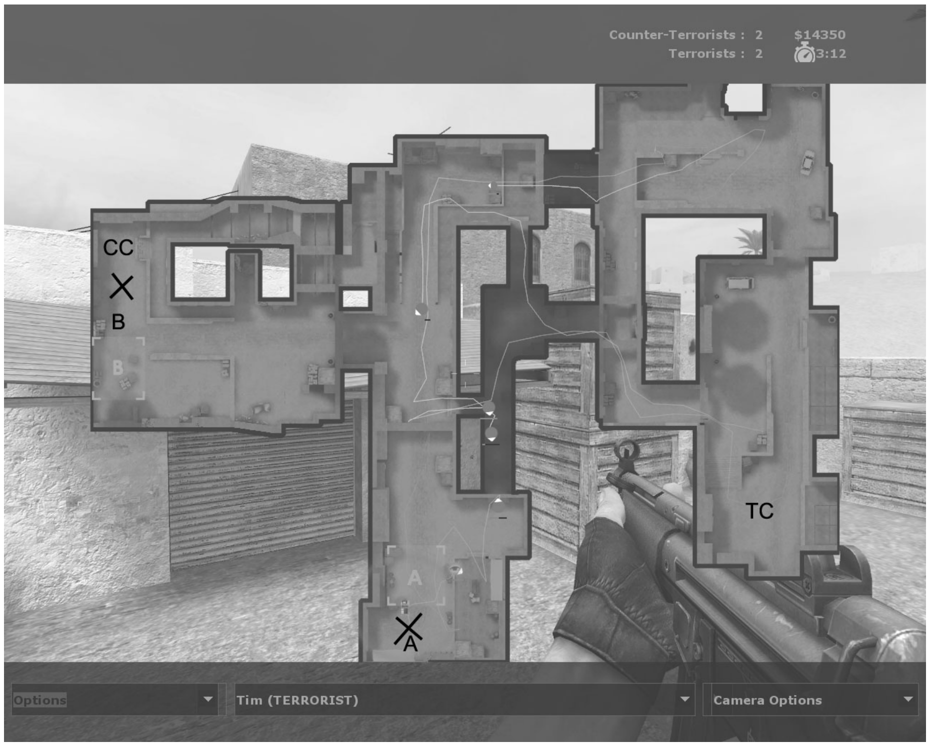


FIGURE 1: Game Map

2.2.5 Counter-strike

Counter-strike is yet another team-based FPS which runs on the Half-life game engine. Counter-strike is one of the most popular open source computer games available in the market, and is played by thousands of players simultaneously on the Internet.

Counter-strike uses bots to simulate human players in teams to give the illusion of playing against actual game players. Bots actually show intelligent behavior in pathfinding, attacking opponents, fleeing away, and planting and defusing bombs. But, they also incorporate the problem of predictability. An experienced game player will be able to predict their behavior and wait for them at strategic locations to kill them. Partly, the reason for this behavior is that the majority of AI in bots is implemented using static finite-state machines (FSM).

We selected the game of Counter-strike as a base case study for our project. Unlike Half-life Counter-strike is a team based game with two teams, namely terrorist and counter-terrorist. The terrorist aims to plant the bomb while counter-terrorist aims to stop them from planting the bomb and killing all of them.

Figure 1 shows a standard map of Counter-strike called DE_DUST. On the map, two sites labeled A, and B are bomb sites where a terrorist aims to plant the bomb. On the contrary, a counter-terrorist aims on defending these bomb sites and if a bomb gets planted by a terrorist, then a counter-terrorist tries to defuse the bomb before it explodes. In the beginning of each round, both the teams are located at designated locations on map. For example, the position labeled CC in figure 1 is ***counter-terrorist camp***, which is the location of counter-terrorists at the beginning of each round. Similarly, the position marked by label TC in figure 1 is the ***terrorist camp*** for terrorists. Once the round begins, they start advancing to different location in map, simultaneously, fighting with each other on encounters and thereby trying to achieve their respective goals.

While we have chosen to use simulation for our experiment, herein, we provided an overview of the game we draw inspiration from. In next the section we will see use of bots which can replace human player and undertake the role of terrorist or counter-terrorist.

2.3 Bots in computer games

Bots in Counter-strike, also called NPCs, are used to replace human players. Bots play as a part of the team and achieve goals similar to humans. Bots simulate human players and are aimed to give game players the 'illusion' of playing against actual human players. Currently, bots used in Counter-strike are programmed to find path, attack opponent players, or run away from the site if they have heavy retaliation, providing an illusion that they are intelligent. Similar species of bots are used in many other FPS games, with similar methods of programming.

Bots are usually pre-programmed according to the requirements of a game and play for or against human players. Based on the method of programming, there can be two styles of bots [11]:

1. Static: Static bots are static in the levels and maps have already been processed. This means that they need to have all information about the maps and level prior to the start of game.

2. Dynamic: Dynamic bots learn as they go through the level. They can be played at any level while static bots cannot.

Both these can techniques produce good quality bots, with a single difference that dynamic bots can learn through levels while static cannot. Usually, bots in computer games are modeled using a FSM as shown in figure 2, where rectangles represent possible states and leading edges show transitions between states. This is just a simplified representation of actual bots, where many more such states exist with more complicated transitions. A FSM for bots is quite self explanatory. First the bot starts by making initial decisions viz. game strategies, buying weapons, etc. and then

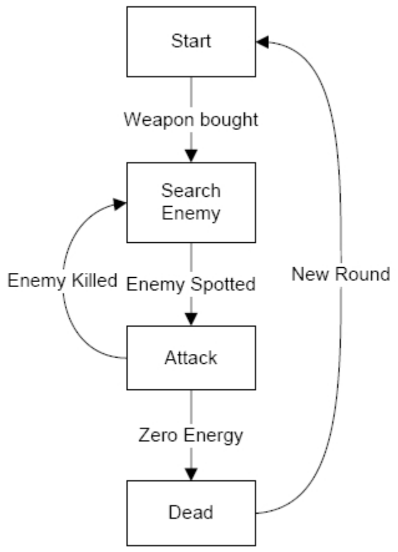


FIGURE 2: A Prototypical FSM for a bots

starts searching for enemies. After the enemy is spotted, it makes a transition to attack state in which she fires bullets at enemy. A Bot may kill an enemy. In that case, it will again start searching for enemies as shown in figure 2. Also, a bot could be in any of the above mentioned states and might get killed by the enemy.

Herein, we have presented a model of bots that are being used in the majority of FPS games. This model inherently has problems. For example, all the transitions shown in FSM are converted to conditional statements. Moreover, conditions of these statements use crisp value to make decision. Precious time is consumed on configuring these crisp values. These motivated us to use reinforcement the learning technique called Q-Learning which can eliminate the use of these crisp values.

2.4 Promising AI methodologies for video games

In the current section we want to explore few of the most promising academic AI techniques that have been implemented in video games, along with their merits, and weaknesses.

2.4.1 A^* Pathfinding

Pathfinding is necessary for almost every video game, for navigating the computer player in a complex environment. Fortunately, an effective solution to this problem is provided by A^* algorithm which is one of the most commonly used algorithms by game programmers [3][4]. The reason A^* is widely used is that it guarantees to find a path between two points if one exits. Also,

A^* is relatively an efficient algorithm which makes it more popular [1].

Formally, A^* is a best-first graph search algorithm to finds the path with least-cost between given initial node and to one goal node using distance-plus-cost heuristic function (f(x)). A^* uses two functions to compute distance-plus-cost function:

- Path-cost-function(g(x)): calculates the cost from start note to current node, and
- Heuristic estimate (h(x)): a heuristic function to estimate the cost from current node to goal node.

So, f(x) = g(x) + h(x) is used to determine a promising path.

Though efficient, sometimes A^* can consume considerable CPU cycles when finding simultaneous path for multiple game agents. So, few optimization techniques have been used to improve the performance of the A* algorithm [5][6]. Fortunately, for most pathfinding problems, A^* is the best choice [1].

2.4.2 Fuzzy Logic

Fuzzy logic is derived from fuzzy set theory. It is a method of dealing with reasoning using approximate values rather than precise crisp values. On the contrary to conventional set theory, fuzzy theory uses membership values in the range between 0 to 1 instead of using 1 and 0. These membership values are used to represent linguistic variables like short, average and tall. The main idea behind fuzzy logic is to represent a computer problem in the way a human being would.

Consider an example of a FPS bot, which has various inputs such as Ammo, Hit points, enemy units, etc. as shown in table 1. Based on these parameters a bot needs to decide whether to Attack or not. A typical membership function for this type of system will be as shown in table 1.

TABLE 1: Fuzzified Bot's Internal State

Linguistic Variable	**Membership Function**
Inputs	
Ammo	Less, Medium, High
HitPoints(HP)	Low, Medium, High
EnemyUnits(EU)	Few, Average, More
FriendlyUnits(FU)	Few, Average, More
DistanceFromCover(DfC)	Small, Medium, Far
Outputs	
Attack	Low, Medium, High

There are various techniques to convert the crisp inputs to and fro from these membership variables. The method to achieve this is called fuzzification. Based on this fuzzified model, FSM for bots(figure 2) can be modeled using simple fuzzy rules as show in table 2.

TABLE 2: Fuzzy Rules

Number	**Rule**
R1	IF ammo is H & HP is H & EU is F THEN attack is H
R2	IF ammo is M & HP is M & EU is A & FU is A THEN attack is H
R3	IF ammo is L & HP is L & EU is F & DoC is S THEN attack is M
R4	IF ammo is L & HP is L & EU is M & FU is F THEN attack is L

This kind of fuzzy system model reduces the time and effort game programmers require for configuring crisp values. Also, the number of rules needed in a bot can be significantly reduced using a fuzzy system.

2.4.3 Artificial Neural Network (AAN)

The artificial neural network (AAN) is a computational model that attempts to model and simulate a biological neural network. AAN is network of nodes that are connected by links. These links have numeric weights attached to them. Weights acts as the primary memory of the network and the network learns by updating these weights. In the network, a few nodes are connected to the environment. These nodes get inputs from the environment and provide output to the same. Weights in the network are modified to generate output desirable to the environment [7].

Again consider an example of a bot that needs to tune parameters such as distance of enemy, energy, number of friendly units, etc. An example of AAN to tune these parameters is shown in figure 3. This is a multilayer feed-forward neural network with distance of enemy, energy, number of friendly units, ammo and type of weapon as input nodes. Based on these inputs, the network returns the action of whether to hide or attack. It requires a training set to adjust its weights so that the neurons fire correctly. This is an offline method of machine learning. Table 3 shows a sample training set which can be provided to this network to adjust its weights. A training set contains set of input and output values. Each set should have four input data for distance of enemy, energy, number of friendly units, ammo in the gun and type of weapon, and of two output values: whether to attack or hide.

An AAN provides a programmer a method to eliminate hard coding of all the vales, which is time consuming. Although efficient in learning, it is offline and often computationally expensive.

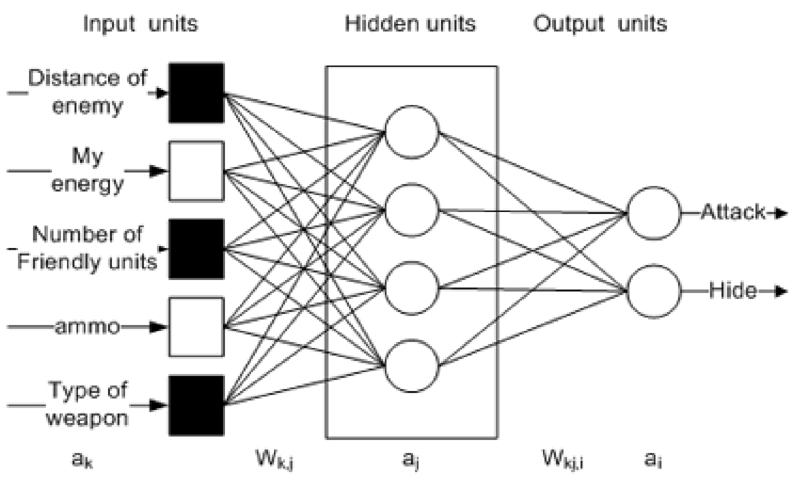


FIGURE 3: AAN for bots

TABLE 3: Training Set

DistanceOfEnemy	Energy	FriendlyUnits	Ammo	TypeOfWeapon	Attack	Hide
0.5	0.3	0.1	0.9	2	0.1	0.0
0.1	0.6	0.2	0.9	5	0.0	0.1
0.2	0.9	0.6	0.7	8	0.1	0.0
0.3	0.1	0.0	0.3	2	0.0	0.1
0.6	0.5	0.1	0.1	4	0.1	0.0
0.2	0.9	0.1	0.1	1	0.1	0.0

2.4.4 Genetic Algorithms

Genetic Algorithms(GAs) are adaptive heuristic search algorithms premised on the evolutionary ideas of natural selection and genetic. The basic concept of GAs is to simulate processes in natural system necessary for evolution, specifically those that follow the principles of survival of the fittest. As such, they represent an intelligent exploitation of a random search within a defined search space to solve a problem. A typical evolution process used in genetic algorithm consists of following four steps[1]:

1. Each individual in first generation represents a possible solution to the problem. Normally, each individual in the population is randomly generated. In game development, however, we know in advance what combinations are likely to produce good solution. It is still important to produce diverse population for the genetic process to be effective. For example, we want the GA to the tune the parameter Attack/Hide based on the input parameter Energy. We can prepare a set of parameters which can be current-state with the anticipated actions:

```
class Agent{
        int DistanceOfEnemy;
        int Energy;
        int FriendlyUnits;
        int Ammo;
        int TypeOfWeapon;
        int Attack;
        int Hide;
}
```

The next step is to generate chromosomes of this set of parameters and initialize them with set of random numbers. This will be our first generation of chromosomes:

```
class GA{
 Agent chromo[populationSize] = new Agent();
 for(int  i=0;i<populationSize;i++){
        chromo[i].DistanceOfEnemy = Random();
        chromo[i].Energy = Random();
        chromo[i].FriendlyUnits = Random();
        chromo[i].Ammo = Random();
        chromo[i].TypeOfWeapon = Random();
        chromo[i].Attack = Random();
        chromo[i].Hide = Random();
  }
}
```

2. The fitness of the each member of the population must be evaluated, so that a particular chromosome can be ranked against another chromosome. This is done with a fitness function, which is the objective function that ranks the optimality of a solution. This generates set of best individual for solving the problem. For our current example, we can simulate the game using the game engine and place our agent in situations represented by the current sets of parameters, and observe the result based on observation from combat which can be used to define the fitness for a particular chromosome.

3. The next step is to choose the individuals whose traits are needed in the next generation. Each chromosome can be ranked based on fitness function using a simple sorting algorithm, and individuals with the best fitness scores are selected.

4. The Final step is to create new individuals form the the set of individuals selected, and combine their chromosomes. This is an evolutionary step, where crossover takes place. In our example, we can pick out parameters from fit chromosomes ($chromosome_j$ and $chromosome_k$) and exchange them to produce new individuals($chromosome_i$) as show below:

```
crossover(int i,int j,int k){
        chromo[i].DistanceOfEnemy = chromo[j].DistanceOfEnemy;
        chromo[i].Energy = chromo[k].Energy;
        chromo[i].FriendlyUnits = chromo[j].FriendlyUnits;
        chromo[i].Ammo = chromo[k].Ammo;
        chromo[i].TypeOfWeapon = chromo[j].TypeOfWeapon;
        chromo[i].Attack = chromo[k].Attack;
        chromo[i].Hide = chromo[j].Hide;
}
```

By repeating the last two steps, we can tune the parameters for the bot. Genetic algorithms can be computationally expensive, even though they has same merits such as saving time to tune up the individual parameters in hardcoded bots' logic.

2.4.5 Q-Learning

R. Sutton at. al. explained reinforcement learning in very simple terms[10]:

> "***Reinforcement learning*** is learning what to do–how to map situations to actions–so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics–trial-and-error search and delayed reward–are the two most important distinguishing features of reinforcement learning".

We are going to use ***Q-learning***, a reinforcement learning technique, that learns an action-value function. This function provides us with the expected utility of pursuing an action in a given state and following a fixed policy afterward. In simpler terms, an agent using Q-learning learns a mapping for which action he should take when he is in one of the states of the environment. This mapping can be viewed as a table, called a ***Q-table***, with rows as states of the agent and columns as all the actions an agent can perform in its environment. Values of each cell in a Q-table signify how favorable an action is given that an agent is in particular state. Therefore, an agent selects the best known action, depending on his current state: $\arg\max_a Q(s,a)$.

Every action taken by an agent affects the environment, which may result in a change of the current state for the agent. Based on his action, the agent gets a reward (a real or natural number) or punishment(a negative reward). These rewards are used by the agent to learn. The goal of an agent is to maximize the total reward which he achieves, by learning the actions which are optimal for each state. Hence, the function which calculates quality of state-action combination is given by :

$$Q : S \times A \to R$$

Initially, random values are set in the Q-table. Thereafter, each time an agent takes an action; a reward is given to agent, which in turn is used to update the values in Q-table. The formula for updating the Q-table is given by:

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{old\ value} + \underbrace{\alpha_t}_{learning\ rate} \times [\overbrace{\underbrace{r_{t+1}}_{reward} + \underbrace{\gamma}_{discount\ factor} \underbrace{\arg\max_a Q(s_{t+1},a_t)}_{max\ future\ reward}}^{expected\ discounted\ reward} - \overbrace{Q(s_t,a_t}^{old\ value}],$$

where r_t is reward at any given time t, α_t is the learning rate and γ is discount factor.

The major advantages of using Q-learning are that it is simple, and it will support dynamic online learning.

2.5 Other AI Techniques

Few other widely used AI techniques in games are[8][9]:

- Decision tree
- Bayesian networks, and
- Flocking

3 RELATED WORK

In our research we considered one of the most commonly used game character: bot, as a case study and used machine learning on improving its behavior A 'bot'(short for robot) is a Non-Player Character(NPC) in a multiplayer video game, that is designed to behave similar to a human-controlled player[11]. Examples are the bots used in games like Counter-Strike[17], Half-Life[18] and Quake[20]. Several bots and their source code are available online to game players and game developers. J. Broome on his website has provided information about development of a bots for the game of Half-Life and had designed several bots for the same[13]. He also explains the creation of 'Half-Life MOD'[2], getting the source code of the game, compiling the code, and finally using it for the bot development or modification. Our main focus for this research is a game called Counter-Strike and its bot, which itself is a MOD for Half-Life and runs on the Half-Life game engine.

N. Cole et. al. argues that to save computation and programmer's time, the game AI uses many hard-coded parameters for bot's logic, which results in usage of enormous amount of time for setting these parameters [15]. Therefore, N. Cole et. al. proposed the use of genetic algorithm for the task of tuning these parameters and showed that these methods resulted in bots which are competitive with bots tuned by a human with expert knowledge of the game. N. Cole at. al. selected the parameters to tune, allowed them to tune while running genetic algorithms, evolved bots against each other, and finally tested these evolved bots against the original bots to test their performance.

Another related work was done by S. Zanetti et. al. who used the bot from the FPS game Quake 3, and demonstrated the use of Feed Forward Multi-Layer Neural Network trained by a Genetic Algorithm to tune the parameters tuned by N. Cole at. al.[16]. Albeit, their resulting bot did not reach the competitive playable level.

Even though there have been attempts to use machine learning techniques for bots, all of used offline learning. But, in remaning section we will demonstrate the use of Q-Learning - an online learning algorithm may prove to be more desirable compared than these offline learning algorithms.

[2]A Half-Life MOD' is a modification that someone has made to the single player or multiplayer version of the game Half-Life. These MODs usually incorporate new weapons, new levels (maps), and/or new methods or rules for playing the game[13].

4 APPROACH

In section 2.3, we had a looked on how bots are being modeled using FSMs, but there are inherent flaws in using classic FSM model for bots. All transitions/rules needs to be hardcoded in bots logic. Hence, the bots appear to show intelligent behavior without using any academic AI techniques.

As all the rules are hardcoded, programmers have to spend time configuring their parameters. For example, a rule for agent's attacking behavior based on agent's speed and energy is shown in algorithm 1. Programmers need to spend considerable amount of time configuring parameters such as energy, distance of enemy, etc., for which they run a large number of simulations, wasting precious man-hours.

Algorithm 1 Hardcoded rules

```
1: if $agent.speed \geq 4\%$ & $agent.range \geq 4\%$ then
2:     attack()
3: else
4:     flee()
5: end if
```

Moreover, even after programmers have spent precious time on configuring these parameters, the bots' behavior will become predictable to an experienced game player. A game player spends hours playing the same game and same level, and therefore reaches a threshold at which he can predict the bots' behavior because they repeatedly perform similar behaviors due their hard-coded logic. This, make a game less interesting to an experienced game player and eventually may lose interest in the game.

To avoid use of hard-coded bots with static behavior, one needs to use online learning techniques, which can adapt bots' behavior to the play of human players. We propose using the reinforcement learning technique called Q-learning for developing sophisticated bots.

4.1 Simulation Environment

For testing Q-learning on bots, we required a simulation environment and efficient bots using this environment. At first, we were motivated by the original source code of the counter-strike game available to us. Counter-strike uses a product call ***Steam***, a content delivery system, to launch the game, configure its various parameters, and play online with other players [25]. After having installed, counter-strike we were able to download the original source code of the game using *Steam*. We were expecting the source code for counter-strike but instead it was the code of half-life, the original game engine on which counter-strike was build. Though we were able to modify and launch the game, it did not have complete source code for bots. We attempted writing our own code for bots from scratch and launch a dummy bot in the game. However, We were not able to improve the bots' code significantly because the original code was not properly documented and was very complex. It became clear that it was infeasible for us to work with this environment for this research project.

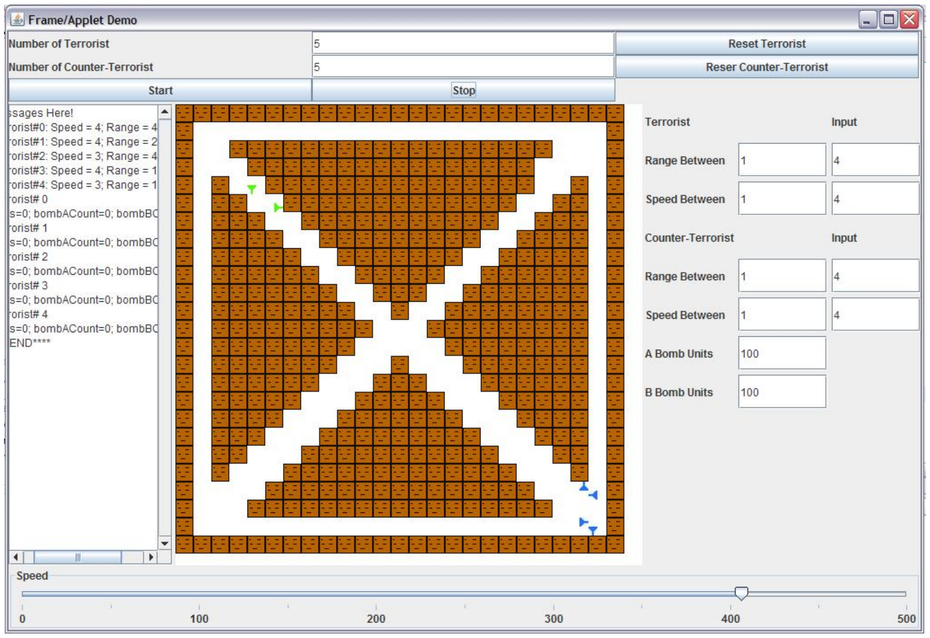
Next, we tried Netlogo, a cross-platform multi-agent programmable modeling environment, for simulation [23]. We were able to build an environment with bots fighting with each other, but Netlogo had its own limitations. We were not able to implement required algorithms given the Netlogo's limitations in integrating with Java or any other programming language.

Finally, we developed a scaled down abstraction of Counter-Strike in Java, and simulated bots in this environment.The miniature version of Counter-Strike is shown in figure 4. Herein, brown colored bricks are visible which form the boundary of the map and act as obstacles for agents. There are two kinds of agents, blue and green, which navigate through the map formed by the bricks. Each of the blue and green agents imitates the behavior of terrorists and counter-terrorist respectively from Counter-Strike. Moreover, there are two sites labeled A and B in figure 5 which are similar to the bomb sites in counter-strike. In figure 5, sites labeled T and C are green and blue base camps respectively. Before the start of a game, we specify the number of each type of agents. The green agents goals will be to plant bomb in one of the sites (either A or B) or kill all blue agents. Blue agents will aim to defend these sites and kill all green agents. This provide us with an environment similar to a classic FPS game, where two autonomous sets of agents fight with each other, but which we can easily experiment with dynamic learning for bots.

Additionally, we have designed the environment in such a way we can provide values for parameters such as range and speed for agents beforehand. Also, we can provide the number of bomb units for bomb being planted on each bomb sites. By bomb unit, we mean number of units needed to plant a bomb. For example, if bomb units are equal to 100 then a single agent will require 100 time steps to plan the bomb. If there is more than one agent trying to plant the bomb, suppose two then both of them together will plan the bomb in 50 time steps. These parameters will be useful in simulating games with agents having different traits.

4.2 Methodology

Our aim is to investigate the Q-learning algorithm for improving the behavior of the green agents, while keeping the blue agent's behavior static.



Number of Terrorist: 5
Number of Counter-Terrorist: 5

```
ssages Here!
orist#0: Speed = 4; Range = 4
orist#1: Speed = 4; Range = 2
orist#2: Speed = 3; Range = 4
orist#3: Speed = 4; Range = 1
orist#4: Speed = 3; Range = 1
orist# 0
s=0; bombACount=0; bombBC
orist# 1
s=0; bombACount=0; bombBC
orist# 2
s=0; bombACount=0; bombBC
orist# 3
s=0; bombACount=0; bombBC
orist# 4
s=0; bombACount=0; bombBC
END****
```

Terrorist		Input
Range Between	1	4
Speed Between	1	4
Counter-Terrorist		**Input**
Range Between	1	4
Speed Between	1	4
A Bomb Units	100	
B Bomb Units	100	

FIGURE 4: Simulation

The static Blue agents run a simple algorithm (Algorithm 2), in which if they spots green agents they shoots a new missile, else they continue moving on map according to plan specified in planC.txt . planC.txt is similar to plan.txt used by green agents; details of which are given due in course.

Algorithm 2 Static Blue Agents

```
1: if $s.hasTerror()$ then
2:     $attack()$
3: else
4:     move according to plan
5: end if
```

For the dynamic green agents, we need to construct a Q-table. The Q-table has a finite number of states an agent can have at any given time, and a finite number of actions an agent can take in these states. Values of each cell in the Q-table, called utilities, shows how favorable an action is given that an agent is in the particular state. Our aim is to use online learning to determine a policy i.e. utility values, so that these bots will learn based on the game-play of a human-player.

If these values are set randomly, then initially bots will behave randomly, which is not desirable because if bot behaves randomly a game player will consider such bot stupid. So, it will be more advantageous to train bots before shipping the actual game. We propose training bots using small numbers of abstract states initially, before shipping the game. The actual game will

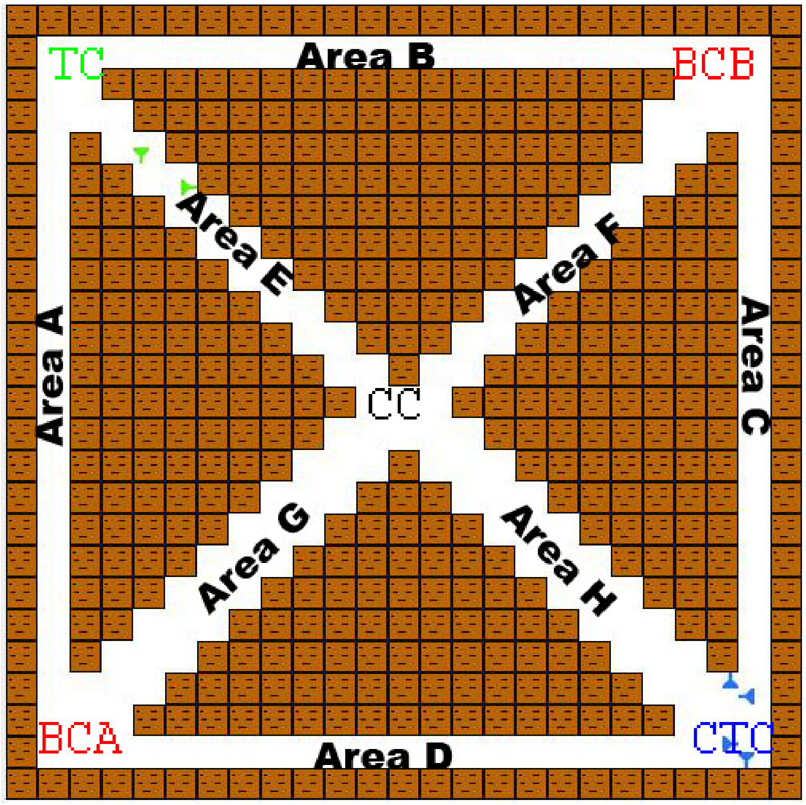


FIGURE 5: MAP divided into Areas

use a large number of more detailed states, which will be the subsets of the states for which bots were initially trained. The speed of Q-learning is influenced by the number of states and actions in the Q-table. Learning with large number of detailed states can be very slow. Hence, the partially trained bots with superset of current states, which are shipped, will not behave randomly and will adapt to the behavior of human, who will notice the change.

Hereby, for example figure 5 shows experimentation in which we have divided the map into 8 areas. For simplicity we plan on starting with eight ***areas*** with which will form the state: set $A = \{A, B, ..., H\}$ for the agents.

Again, agents will select randomly one of the six plan specified in *plan.txt*. A sample plan.txt is shown below:

$0: \ CTC; BCA; BCB; TC; CC;$

$1: \ BCB; BCA; CC; CTC; X;$

$2: \ BCA; BCB; TC; CTC; X;$

$3: \ BCA; BCB; CTC; X; X;$

$4: \ CC; BCB; CTC; X; X;$

5 : $TC; CC; CTC; BCA; BCB;$

In plan.txt, each possible location mnemonics is separated by semi-colon. Each mnemonics signifies a location on map as shown in figure 5. For example, plan - CTC;BCA;BCB;CTC;CC; instructs agent to navigate in sequence as following: $CTC(Counterterroristcamp) \rightarrow BCA(bombsiteA) \rightarrow BCB(BombsideB) \rightarrow TC(TerroristCamp) \rightarrow CC(CentralSite)$ So, agents' second state is ***plans***: set $P = \{0, 1, ..., 5\}$ of size six.

In addition, we will use ***enemy present*** of size two as state: set $E = \{0, 1\}$, which signifies whether opponent are present in individuals range(0) or not(1). Hence, we have state space of 96 states ($A \times P \times E$). An agent, who will be in one of these states at any period in time, will perform one of the following actions: Attack or Ignore.

We do not have any immediate method of predicting agents' future state and determining whether an action currently execute is fruitful or not (rewards). We need to wait for agent's next turn to determine rewards. Hence, we update state s_t value when agent is in state s_{t+1}. In state s_{t+1} we can determine the rewards for action of agent in state s_t and also have knowledge of its future state i.e. s_{t+1}. Again, if suppose an agent fired a missile then we cannot determine the rewards until state s_{t+x}, where $x > 1$, is the state when missile actually hits a blue agent or defuse without hitting anyone. In such a scenario, we ignore the intermediate states between state s_{t+x} and s_t, and directly update values of state s_t based on values of state s_{t+x}.

Initially, in order to chose any action while an agent is in particular state we used utilities as probabilities. But this algorithm was not generating better bots. So we used an 'exploration rate(ϵ)' as probability for choosing the best action. Suppose, if the exploration rate of agent is 0.2, then an agent will choose action with maximum utility value with probability of 0.8 and any other actions with probability of 0.2. Usually, low exploration rates, between 0.0 to 0.3, are used. Therefore, an agent selects an action with maximum utility most of the time and ϵ determines the probability of exploring other actions.

Algorithm 3 Dynamic Green Agents

```
1: $currentState = getCurrentState()$
2: $prevState = getPreviousSate()$
3: $action = selectAction(currentState)$
4: if action = 0 then
5:     $attack()$
6: else
7:     $ignore()$
8: end if
9: $updateQtable(prevState, currentState, rewards)$
10: $setPreviousState(currentState)$
```

Algorithm 4 *selectAction(currentState)*

```
if QTable[currentState][0] > QTable[currentState][1] then
    $prob0 = 1 - \epsilon$
else
    $prob0 = \epsilon$
end if
$rand$ = random number between 1 to 100
if $rand < prob0 * 100$ then
    action = 0
else
    action = 1
end if
return action
```

Algorithms 3 and 4 summarize the algorithms used by green agent wherein it is learning the best action i.e. attack or ignore(0 or 1). Algorithm 4 is quite self-explanatory: return an action based on exploration rate (ϵ). In algorithm 3, during the first two steps agent retrieves its current state and previous states. Then the agent selects an action based on its current state. Next, if the action is 0(mnemonics for attack action), then agent shoots a missile, else it just continues according to its plan. Finally, the agent updates its Q-table based on current and previous state, and stores the current state as previous state to use for the next iteration.

Hereby, we have raw material for our experimentation i.e. using Q-learning for bots. Q-learning algorithm will learn action-value function for state space and actions, based on which bots' behavior will evolve.

5 EXPERIMENT AND RESULTS

We investigate designing agents using Q-learning which learned different behaviors based on rewards they are getting.

Key issues with any learning techniques is setting various parameters, which in case of Q-learning are learning rate(α), discount factor(γ), and exploration rate(ϵ) . Learning rate determines the extent to which recently gained utilities should override previously learned utilities. Therefore, a learning rate of 0 means agents are not learning and 1 means that agents will consider most recent utility only. Discount factor determines the importance of future rewards. A high discount factor means agents give greater importance to future rewards, while a low discount factor means agents are opportunistic, giving importance to current rewards. Agents select an action with greater utility value with probability of 1 - ϵ and other action with probability of ϵ . Exploration rate being the probability of trying new actions, high exploration rate means agents will try action other then action with greater utility more often.

Provided a flexible simulation environment, inspired from environment in the Counter-strike game, varieties of experiments are possible. Albeit, bots in Counter-strike, as with most other FPS games, need to learn basic behaviors such as combat and planting the bomb. Therefore, we experimented with rewards function in order for bots to learn these basic behaviors i.e. learning to fight and plant the bomb. Apart from this, a model of a actual game will have a large number of states. Moreover, as the number of states grows in Q-learning the size of the Q-table grows; simultaneously slowing down the speed of learning. Hence, we propose to train the bots with a small number of abstract states which are a superset of the more detailed states used by actual game. Finally, these learned utility values are distributed among large number of detailed states in the actual game and an agent continue online learning thereafter.

Evaluation of these agents is based on the maximum fitness green agents would reach against static blue agents. In all the experiments, fitness of agents is measured by the ratio of number of rounds won by green agents against the number of round won by blue agents. By round, we mean a single game cycle, wherein either green agents won by killing all the blue agents or planting the bomb, else blue agents won the round by killing all the green agents. For each experiment we modified reward function so that agents can learn different. The remaining section provide detailed experimental setup and results.

TABLE 4: Experimental Results for Learning to Fight

Learning Rate (α)	**Discount Factor (γ)**	**Exploration rate (ϵ)**	**Fitness**
0.10	0.90	0.10	0.6471
0.25	0.90	0.10	0.6809
0.30	0.95	0.10	0.8536
0.40	0.95	0.10	0.8533
0.50	0.90	0.10	0.8818
0.50	0.97	0.10	0.9372
0.50	0.98	0.10	0.9875
0.50	0.99	0.10	1.0065
0.50	1.00	0.10	0.6838
0.55	0.99	0.10	1.0620
0.56	0.99	0.10	1.0620
0.60	0.95	0.10	0.8500
0.60	0.99	0.10	0.9859

5.1 Learning to fight

For the first experiment, we wanted to train the agents to learn combat behavior. Agents had only two actions to choose from: Attack or Ignore opponents. In the attack action, agents shoot a missile, while in the ignore action agents just ignore the presence of a nearby enemy and continue their current plan.

In order for agents to learn that shooting a missile is costly, if it is not going to be effective, we gave small negative reward of -0.1 if agent shoots a missile. If the agent gets hit by an enemy missile, the agent gets a small negative reward of -0.2. Agents were given a large positive reward of +10 if they kill an enemy agent. All the values of Q-table were set to zero before training. Table 4 shows the result of various combinations of α , γ and, ϵ and the fitness level the agents reached.

Figure 6 shows the learning curve for three different combination of α , γ , and, ϵ . Similar curves are also observed for remaining combination of parameter. Unexpectedly, the combination with $\alpha = 0.56$, $\gamma = 0.99$ and, $\epsilon = 0.1$ produced agents with maximum fitness. A high value of γ signifies that future states are playing an important role in determining the selection of current actions. Reinforcement learning technique tend to produce curves with high fluctuations if learning rate is high. But, in our experiment we observed a very steady learning curve, as seen in figure 6. Exploration rate of 0.1 is normal for this type of experiments. Notice that the curve crosses the fitness level of 1.0 around 3000 rounds and then curve becomes steady showing very little improvement and reaching an asymptote of 1.0620. Fitness value greater than 1 here means that agents are outperforming static agents.

Hence, with this experiment we were able to evolve agents which successfully learned a

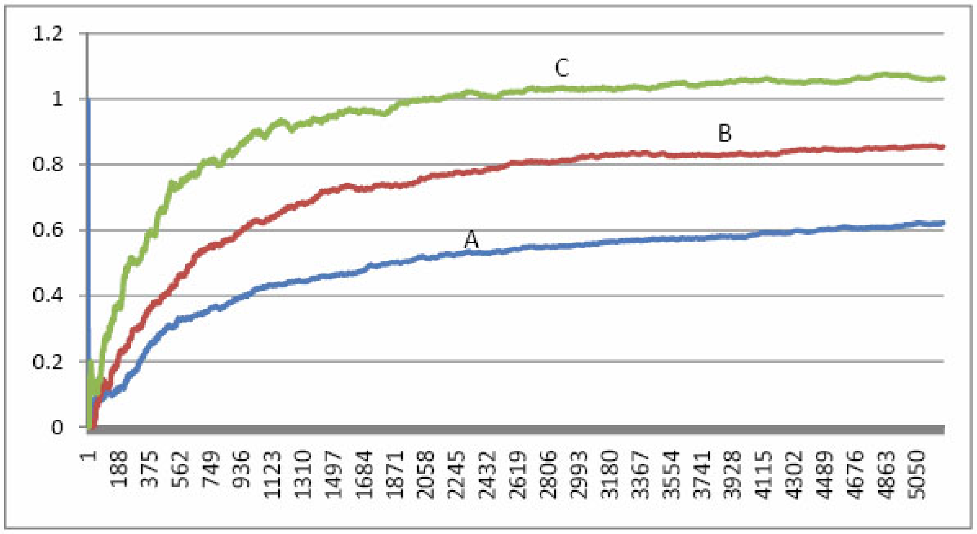


FIGURE 6: LEARNING CURVES FOR LEARNING TO FIGHT: (A) $\alpha = 0.10$, $\gamma = 0.90$ AND, $\epsilon = 0.1$, (B) $\alpha = 0.30$, $\gamma = 0.95$ AND, $\epsilon = 0.1$, AND (C) $\alpha = 0.56$, $\gamma = 0.99$ AND, $\epsilon = 0.1$

combat behavior.

5.2 Learning to plant the bomb

Next experiment was to train the agents for planting the bomb. In the first experiment, we used expert knowledge that killing opponents is good. Now, we want to explore whether bots can evolve to learn that behavior (or better) if they are focused on planting the bomb. Again, agents had two actions to choose from: attack and ignore. But, now planting the bomb action was part of ignore action. So, an agent would plant the bomb if it is in one of the bomb sites as a part of ignore action, else it will continue on its current plan.

Agents did not receive any rewards for killing enemy agents. Instead, rewards are given only when an agent makes attempt to plant the bomb: +4 reward for each unit of bomb planted. Similar to the first experiment, agents were given -0.1 rewards for shooting a missile and -0.2 rewards for being hit by an enemy missile. Also, all the values of Q-table were set to zero before training. Table 5 shows results for various combination of learning rate and exploration rate.

Notice that the best fitness ratio is for the combination of $\alpha = 0.94$, $\gamma = 0.99$ and, $\epsilon = 0.20$. Again, we acquired a high learning rate though the graph shown in figure 7 is quite smooth. We got the same discount rate as before. Herein, we can see that we acquired high exploration rate due the fact that we are not giving any rewards for killing opponent agents. Yet, in order to successfully plant the bomb an agent has to kill the opponent agents otherwise it will get killed by

TABLE 5: Experimental Results for Learning to Plant the Bomb

Learning Rate (α)	**Discount Factor (γ)**	**Exploration rate (ϵ)**	**Fitness**
0.10	0.99	0.10	0.4332
0.20	0.99	0.10	0.3711
0.30	0.99	0.10	0.3745
0.40	0.99	0.10	0.4276
0.50	0.99	0.10	0.4595
0.60	0.99	0.10	0.5405
0.70	0.99	0.10	0.5894
0.80	0.99	0.10	0.6808
0.90	0.99	0.10	0.7453
0.93	0.99	0.10	0.7570
0.94	0.99	0.10	0.7609
0.94	0.99	0.15	0.9849
0.94	0.99	0.19	1.2001
0.94	0.99	0.20	1.2501
0.94	0.99	0.21	1.2092
0.94	0.99	0.25	1.1915
0.95	0.99	0.10	0.7184

them. So, in order learn to kill blue agents it should actually fire a missile more often than in the previous experiment. The utilities require more time to propagate than before because the only location agents are getting positive rewards are in the two corners. Hence, it can be seen in figure 7 that we are required to run this simulation for more number of rounds.

Table 6 shows the Q-values learned from the experiment. A state in the table is represented by a triplet $[PAE]$ where $P = 0, 1, ..., 6$ is the plan number, $A = A, B, C, ..., H$ is the area and, $E = p\ if\ enemy\ is\ present\ or\ n\ if\ enemy\ not\ present$. Values in remaining two columns: attack and ignore are the utility value of taking a particular action. Agent selects the action with greater utility value with probability of 1-ϵ else selects the other section.

Few rows in the Q-table have value 0 or very small value like -0.1 (states: (0 H N), (2 A P), (4 C p), etc.). These are states where agents were not trained because agents rarely used these areas while using a particular plan. Similar is the case with all the states of plan 4 and 2(states: (4 X X) and (2 X X)) because as the both sets of agents(green and blue) are using static plan.txt, green agent while using plan 4 and 2 never encounters the enemy agent. Therefore, all the enemy present state (states: (4 X p) and (2 X p)) are having values zero. Remaining states have values -0.10 because every time a agent shoots a missile, it receives -0.1 reward. We can infer from the table that, for the majority of remaining states, agents learned the following two behaviors:

- To ignore or plant the bomb if enemy is not present. There is no need to shoot a missile if no enemy is present i.e. utility value of ignore action is greater than attack action, for example,

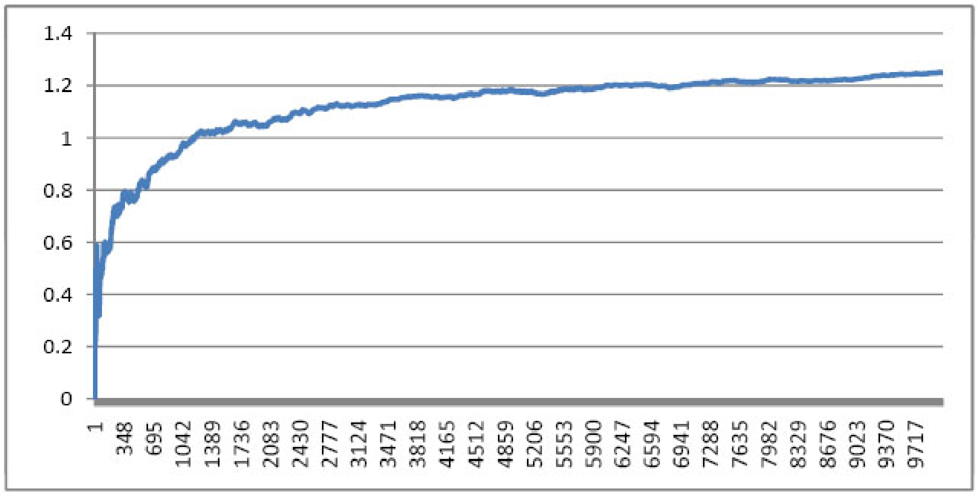


FIGURE 7: Learning Curve for Learning to Plant the bomb

TABLE 6: Q-table for 8 areas

State	Attack	Ignore	State	Attack	Ignore	State	Attack	Ignore
0 A n	14.72	19.33	2 A n	-0.10	0.00	4 A n	-0.10	0.00
0 A p	2.10	1.83	2 A p	-0.10	0.00	4 A p	0.00	0.00
0 B n	26.89	35.14	2 B n	0.00	0.00	4 B n	-0.10	0.00
0 B p	41.38	32.92	2 B p	0.00	0.00	4 B p	-0.10	0.00
0 C n	33.67	33.65	2 C n	0.00	0.00	4 C n	-0.10	0.00
0 C p	41.02	33.10	2 C p	0.00	0.00	4 C p	-0.10	0.00
0 D n	1.25	1.19	2 D n	0.00	0.00	4 D n	0.00	0.00
0 D p	-0.09	1.89	2 D p	0.00	0.00	4 D p	0.00	0.00
0 E n	18.54	18.47	2 E n	0.00	0.00	4 E n	-0.10	0.00
0 E p	34.33	23.51	2 E p	0.00	0.00	4 E p	-0.10	0.00
0 F n	19.58	19.57	2 F n	0.00	0.00	4 F n	-0.10	0.00
0 F p	19.99	19.98	2 F p	0.00	0.00	4 F p	-0.10	0.00
0 G n	1.47	1.58	2 G n	0.00	0.00	4 G n	0.00	0.00
0 G p	2.26	1.82	2 G p	0.00	0.00	4 G p	0.00	0.00
0 H n	0.00	0.00	2 H n	0.00	0.00	4 H n	0.00	0.00
0 H p	0.00	0.00	2 H p	0.00	0.00	4 H p	0.00	0.00
1 A n	507.46	405.38	3 A n	736.73	853.34	5 A n	809.49	479.73
1 A p	424.37	451.57	3 A p	0.00	0.00	5 A p	0.00	0.00
1 B n	0.00	0.00	3 B n	0.00	0.00	5 B n	1282.73	1287.09
1 B p	0.00	0.00	3 B p	0.00	0.00	5 B p	0.00	0.00
1 C n	426.63	446.64	3 C n	808.39	808.71	5 C n	1281.39	1298.65
1 C p	0.00	0.00	3 C p	1272.05	1286.09	5 C p	1001.68	1012.48
1 D n	457.14	444.57	3 D n	816.25	1073.53	5 D n	0.00	0.00
1 D p	341.75	235.38	3 D p	0.00	0.00	5 D p	0.00	0.00
1 E n	346.24	414.05	3 E n	631.78	853.77	5 E n	333.55	333.74
1 E p	508.15	254.64	3 E p	659.66	620.04	5 E p	897.07	1184.53
1 F n	0.00	0.00	3 F n	0.00	0.00	5 F n	726.30	1249.16
1 F p	0.00	0.00	3 F p	0.00	0.00	5 F p	956.68	7948.27
1 G n	387.39	490.14	3 G n	868.36	825.27	5 G n	0.00	0.00
1 G p	501.48	457.98	3 G p	696.26	777.09	5 G p	0.00	0.00
1 H n	325.27	515.06	3 H n	602.72	884.51	5 H n	0.00	0.00
1 H p	608.08	528.55	3 H p	908.80	775.18	5 H p	0.00	0.00

states: (0 A n), (1 E n), (3 A n), etc..

- To attack if enemy is present. An agent learned to attack even though it is not getting any rewards for doing so i.e. utility value for attack action if greater then utility of ignore action if enemy is present, for example, states: (0 A p), (1 D p), (3 H p), etc..

We observe the second behavior due to the fact that rewards propagate from the state where agents plant the bomb to the state where agents shoot a missile. Also, there is a small difference in utility values of both the action in the majority of the states because the same rewards(for bomb planting) also propagate for ignore action. But as agents are able to plant more bomb units only if they killed an enemy agent in a previous state and hence, indirectly learned that killing is required to plant the bomb. Nevertheless, there are few states where even if enemy is present and utilities for ignore state are greater, for example, states: (0 D p), (3 C p), (3 G p), etc. These are the states with areas away from bomb sites so propagation for rewards might require more training or ignore action might actually be a better option to choose(to run away) because the ultimate goal is to plant the bomb.

Bots generated with this experiment outperformed the static bots and learned to attack even though they are not receiving direct incentives. However, we cannot compare the results of experiments in section 5.1 i.e. learning to fight with this experiment. In current experiment i.e. learning to plant the bomb, agents has definite goal of planting the bomb, so we modified plan.txt to achieve this behavior such that final location of each plan is one of the bomb sites (BCA or BCB). This affects the outcome of the game because at first agents were moving randomly, but now they have a definite goal of going to a particular location and planting the bomb.

5.3 Learning for deployment

Above experiments showed that with this technique we are able to generate a competitive bots but, bots with random initial values cannot be supplied with actual games. So bots need to be partially trained before they are actually supplied with a game. Also, training with more number of states, as in the case with actual game, also takes considerably more amount of time. So in this experiment we trained bots for a small number of rounds with agent having fewer states and then used those Q-values to train the agents with large number of states. Rewards and other settings for the experiment is kept similar to experiment is section 4.1 and used combination $\alpha = 0.94$, γ =0.99 and, $\epsilon = 0.20$ for experimentation which produced bots with maximum fitness.

Until now, in all the experiments we have divided the map into 8 areas. We divided our map

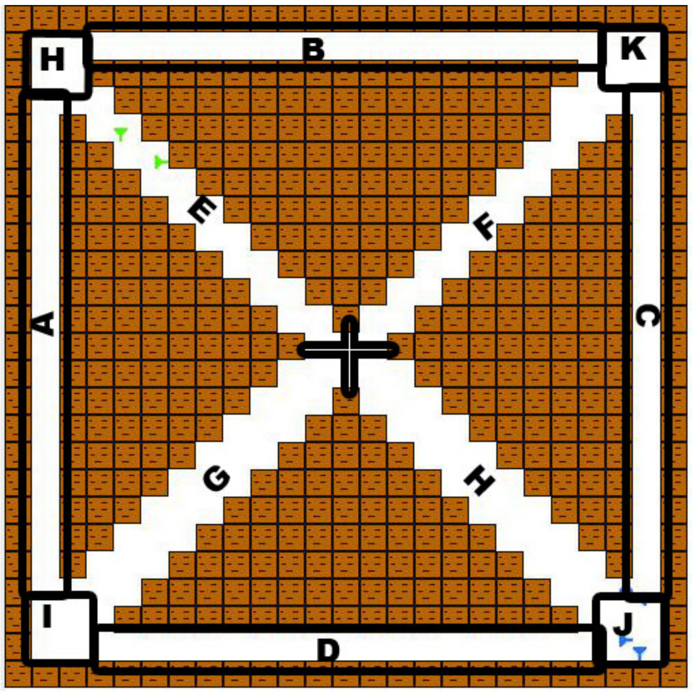


FIGURE 8: 12 Areas

TABLE 7: Experimental Results for Learning for Deployment

Round in initial training with 8 area	Fitness with 12 areas	Fitness with 16 area
0	1.7488	1.6887
500	1.8299	1.7163
1000	1.8023	1.6931
1500	1.8327	1.7306
2000	1.7823	1.7337

first into 12 areas and then into 16 areas. For a map divided into 8 areas size of the Q-table was 96 which got increased to 144 for 12 areas and 192 for 16 areas. Figure 8 and Figure 9 shows the divided map for 12 areas and 16 areas respectively.

Agents are trained on map with 8 areas for 500, 1000, 1500, and 2000 rounds initially and utilities for these agents are stored. These stored utilities are then used as initial utilities for agents to be trained on map with 12 and 16 areas. Here, numbers of states for agents with 12 and 16 areas are more than the agents with 8 areas. Therefore, the utilities for new states are set equal to utilities of old states from which they are generated. For example, area I in figure 8 was part of area A in figure 5 so, utilities of all the states with area I is set equal to utilities of state with area A.

Table 7 shows the highest ratio achieved by agents in each setup, i.e. for 12 and 16 areas. We have almost same values for different initial training which signifies that number of initial training does not play a significant role in determining agent's ultimate performance. The interesting fact

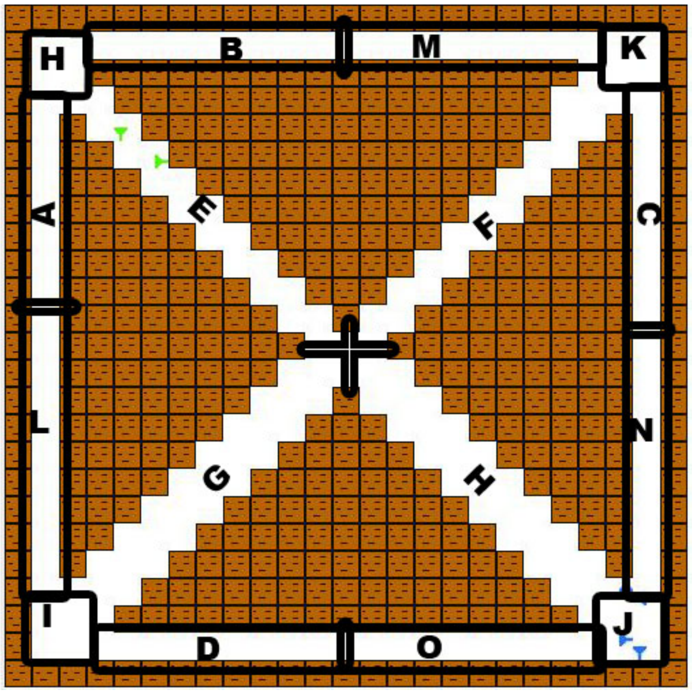


FIGURE 9: 16 Areas

about this experiment is visible in graphs of figure 10 and figure 11. Both the figure shows the comparison graph between learning curve of agents with 0 initial utility values (A) and utilities from trained samples for 500 or 1000 rounds with 8 areas as initial utility values(B). Though both the graph almost converge at the end; notice that initial fitness of the agents with initial training is high and remains high throughout. This shows that initial training provided to bots with fewer states is useful and agents make sudden jumps to certain fitness levels and remain at those level with minor increment. The initial ups and downs seen in both graphs are due to the fact that our evaluation criteria is ratio of green wins verses blue win which keeps on fluctuating due to less samples.

Finally, table 8 shows few selected samples from the Q-table before and after training in 12 areas for 20000 rounds. Before training, utility values of attack action, when an enemy was present, was less than value of ignore action. But after training agent's utility value of attack action became greater than value ignore action. Here, agents evolved to learn to shoot missiles at opponents when one is present. This demonstrated that an agent is capable of learning better and different actions than the initial utilities supplied from small number of abstract states.

We can conclude from this experiment that when partially learned values from abstract states used as initial value for detailed states provided a fitness boost to the agents. Agents thereafter remained at competitive fitness level against static agents and continue learning a better behavior.

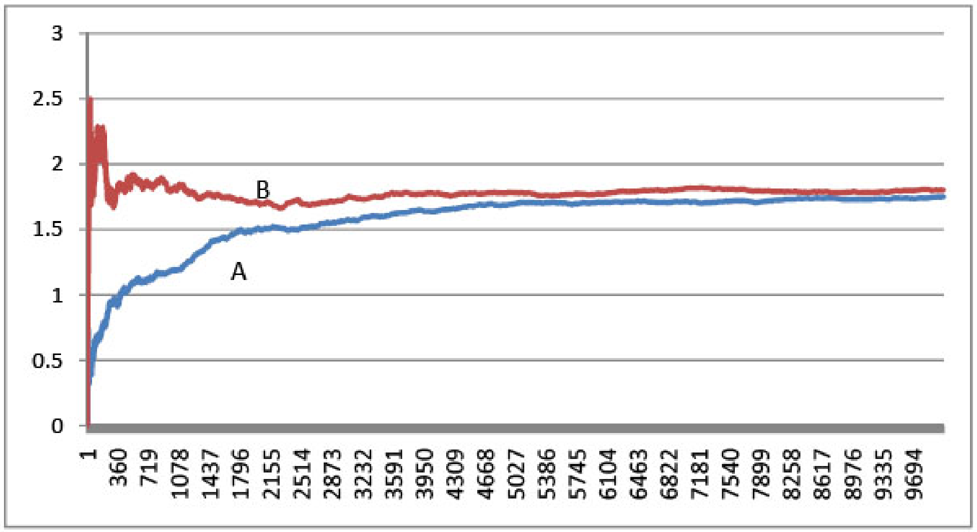


FIGURE 10: Learning Curve with 12 areas, where (A) is the curve without any initial training and, (B) is the curve with 1000 initial training from 8 states

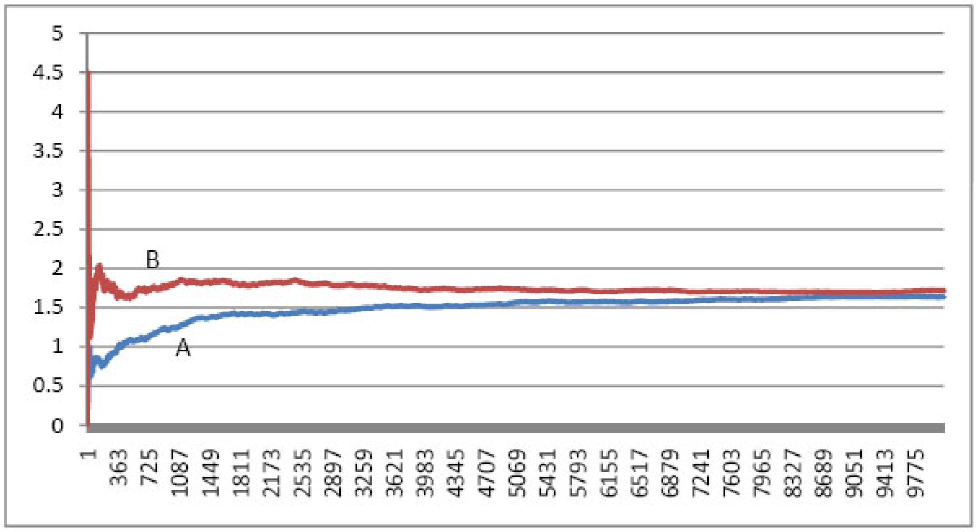


FIGURE 11: Learning Curve with 16 areas, where (A) is the curve without any initial training and, (B) is the curve with 500 initial training from 8 states

TABLE 8: Comparison of Q-Values

State	**Initial Attack**	**Initial Ignore**	**Final Attack**	**Final Ignore**
1 G p	283.44	313.69	940.71	886.04834
1 H p	0	288.57	997.46	916.6944
1 A p	318.27	327.12	1027.06	936.54224
3 C p	378.20	449.48	1343.67	1334.412

6 SUMMARY AND FUTURE WORK

Everyone is fond of playing video games; so are we. Counter-strike was one of the games we were passionate about and have played it for hours at a stretch killing terrorists. So, we were able to figure out the flaws in its game AI. The same was the case with Microsoft's Age of Empires, in which we built kingdoms and fought wars with enemies. This led to further studies of games such as, Black and White, Creature, and Half-life to gain knowledge of game AI used in them. Along with this, we studied the requirements of game AI. Goals of game AI are different than AI techniques studied in academics, yet both of them face similar challenges. We found that very few academic AI techniques are being used by video game programmers.

Although there were a few games that use promising academic AI techniques, most games have hard-coded AI in games logic, making their non-player characters static. As a result, an experienced game player is able to anticipate the behavior of the game's characters. So, as game AI is not competitive, game play becomes boring to human player. Moreover, game programmers are spending a considerable amount of time building these static artificial characters. Hence, we felt a need of learning techniques through which game characters adapt to human players' game play and saves time for game programmers. In order to evolve with human players' game play, it is necessary to use an online learning technique. Also, it is necessary that the algorithm is efficient, and does not consume substantial memory. Hence, we selected Q-Learning, a reinforcement learning algorithm, to evolve bots. We developed a miniature simulation of Counter-strike in Java, for use as the framework for our experiments.

At first, bots were trained in combat. They were given large positive rewards for killing enemy agents and very small negative rewards if they shoot a missile or get hit by enemy missile. In about 5000 rounds with learning rate (α) = 0.56, discount factor (γ) = 0.99, and exploration rate (ϵ) = 0.9 they reached a fitness of 1.0620. This means that they were outperforming their static counter agents and proving Q-learning approach could be effective for this application.

In the second set of experiments, bots were trained to plant the bomb, not concentrating on combat. Bots were not given any rewards for killing enemy agents, but were given rewards only when they plant the bomb. They reached fitness of 1.2501 with $\alpha = 0.94$, $\gamma = 0.99$ and $\epsilon = 0.8$ after 10,000 rounds. Again, we saw that the learned agents were able to outperform their static opponents. While these experiments took more cycles for the agents to reach their terminal performance, they became competitive (performance ratio of 1.0) against the static agents long before this. An interesting result from this experiment is that even though agents are not receiving any rewards for killing opponents, they still learned to do so, because in order to plant the bomb they need to kill enemy agents, else enemy agent will kill them. This validates the appropriateness of our combat training rewards.

Finally, when it comes to use of these bots after releasing the game, bots with random or zero initial utility values cannot be shipped because they will behave randomly. Therefore, for our third set of experiments, we took the Q-tables from the bots trained with a small number of abstract states and used this information as the starting point for bots with a larger number of more detailed states. The resultant bots directly jumped to a competitive fitness level, i.e. above 1 from the beginning. Also, they maintained this level of fitness for 10,000 rounds or more.

It is evident from these results that evolved bots are able to outperform their static counterparts. Moreover, by using the Q-learning algorithm bots were able to learn various behaviors based on the rewards they are getting. Also, having trained these bots with small number of abstract states, we are able to generate a competitive bot for a large number of detailed states. In this learning-based approach, bots learned to attack or ignore the opponents based on their current state, which comprises of location, plan, and enemy present or not. No hard coding of any parameters was required for bots. Bots selected action based on its utility values which was updated dynamically. Hence, by using this approach we can not only reduce the efforts to engineer the hard-coded bots, but also evolve them dynamically to improve their behavior and adapt to human player strategies.

Furthermore, we may still improve the performance of agents by devising a method through which bots can learn to select plans. Currently, we are selecting a plan randomly after each round, but instead we can select a plan based on previous experience of bots. Bots need to select a plan which in past has been proved to be most fruitful. This behavior can be achieved by interpreting their current utilities for using a particular plan. Along with this, we can also use confluence of various learning technique to improve the learning speed of agents. For example, after each round we can use the genetic algorithm to mutate utility values in the Q-table among these agents in order to generate a better population. Currently, all the five agents are learning separately without any intaraction among one another.

Most importantly, we need to test this approach in real simulation of the game, which was our initial attempt. Until then, we cannot judge the actual performance of these agents. Ultimately, bots need to play against human players. Though we have tested these bots against static bots, yet human behavior is very unpredictable.

Nevertheless, there is strong need for convergence of game developers with academic AI community. Academic AI community is indulged in designing various algorithms which can prove advantageous for game developers. Additionally, game developers everyday efficiently simulate real world environment, providing academic AI community ready testbed for testing its newly invented algorithms and AI entities.

BIBLIOGRAPHY

[1] D.M. Bourg and G. Seemann. AI for Game Developers. *O'Reilly Media, Inc., 2004*

[2] B. Scott. AI Game Programming Wisdom by Steve Rabin. *Charles River Media, Inc., 2002, pages 16-20*

[3] J. Matthews. Basic A^* Pathfinding Made Simple. AI Game Programming Wisdom by Steve Rabin. *Charles River Media, Inc., 2002, pages 105-113*

[4] D. Higgins. Generic A^* Pathfinding. AI Game Programming Wisdom by Steve Rabin. *Charles River Media, Inc., 2002, pages 122-132*

[5] D. Higgins. How to Achieve Lightning-Fast A^* AI Game Programming Wisdom by Steve Rabin. *Charles River Media, Inc., 2002, pages 133-144*

[6] T. Cain. Practical Optimization for A^* path Generation AI Game Programming Wisdom by Steve Rabin. *Charles River Media, Inc., 2002, pages 146-152*

[7] S. Russell and P. Norvig. Artificial Intelligent A Modern Approach. *Prentice-Hall, Inc. 1995*

[8] D. Johnson and J. Wiles. Computer Game with Intelligence. *Australian Journal of Intelligent Information Processing Systems, 7, 61-68*

[9] S. Yildirim and S.B. Stene. A Survey on the Need and Use of AI in Game Agents. *In Proceedings of the 2008 Spring simulation multiconference, 2008, pages 124-131*

[10] R. Sutton and A. Barto. Reinforcement Learning:An Introduction. *The MIT Press Cambridge, Massachusetts London, England*

[11] Valve Developer Community, viewed 5^{th} *oct* 2008. http://developer.valvesoftware.com/wiki/Bots

[12] THE BOT FAQ, viewed 5^{th} *oct* 2008. http://members.cox.net/randar/botfaq.html

[13] J. Broome. Botman's bots Half-Life bot development. http://botman.planethalflife.gamespy.com/index.shtml

[14] A. Nareyek. Intelligent Agent for Computer Games. *In Proceedings of the Second International Conference on Computers and Games, 2000*

[15] N. Cole, S. J. Louis, and C. Miles. Using a Genetic Algorithm to Tune First-Person Shooter Bot. *In Proceedings of the International Congress on Evolutionary Computation, 2004*

[16] S. Zanetti and A. Rhalibi. Machine Learning Techniques for FPS in Q3. *ACE'04, June 3-5. 2004, Singapore*

[17] Valve Corporation. Counter-Strike: Source. www.counter-strike.net/

[18] Valve Corporation. Half-Life II. http://orange.half-life2.com/

[19] Gamebots. http://gamebots.planetunreal.gamespy.com/index.html

[20] Id Software, Inc. Quake II. http://www.idsoftware.com/games/quake/quake2/

[21] Lionhead Studios. Black and White. http://www.lionhead.com/BW/Default.aspx

[22] Gameware Development Ltd. Creatures. http://www.gamewaredevelopment.co.uk/creatures_index.php

[23] U. Wilensky. NetLogo. http://ccl.northwestern.edu/netlogo. Center for Connected Learning and Computer-Based Modeling. Northwestern University, Evanston, IL, 1999.

[24] Dictionary.com, turing test, in The American Heritage New Dictionary of Cultural Literacy, Third Edition. Source location: Houghton Mifflin Company, 2005. http://dictionary.reference.com/browse/turing test. Available: http://dictionary.reference.com. viewed: October 27, 2009.

[25] Valve Corporation. Steam. http://store.steampowered.com/

VITA

Graduate School

Southern Illinois University

Purvag G. Patel Date of Birth: September 16, 1985

800 E Grand Ave APT#17G, Carbondale, IL 62901

4 Anuradha Apt, Vasana, Ahmedabad, Gujarat 380007

purvag2003@gmail.com

Gujarat University

Bachelor of Engineering, Coputer Engineering, June 2007

Thesis Title:

Improving Computer Game bots' behavior using Q-Learning

Major Professor: Norman Carver

Publications:

Purvag Patel and Henry Hexmoor (2009). Designing BOTs with BDI Agents at The 2009 International Symposium on Collaborative Technologies and Systems.