Tuning Computer Gaming Agents using Q-Learning

Purvag G. Patel
Department of Computer Science
Southern Illinois University Carbondale
Carbondale, IL 62901
Email: purvag@siu.edu

Norman Carver
Department of Computer Science
Southern Illinois University Carbondale
Carbondale, IL 62901
Email: carver@cs.siu.edu

Shahram Rahimi
Department of Computer Science
Southern Illinois University Carbondale
Carbondale, IL 62901
Email: rahimi@cs.siu.edu

***Abstract*—The aim of intelligent techniques, termed *game AI*, used in computer video 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. 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 known to be intractable if they use a detailed model of the world, and also require tuning of various parameters to give satisfactory performance. Therefore, this paper 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.**

I. Introduction

SINCE the advent of game development, game developers have always used game AI for developing the game characters that could appear intelligent. All the games incorporate some form of *game AI*. It can be in the form of ghosts in the classic game of PAC man or sophisticated bots in first-person shooter(FPS) games such as Counter-Strike and Half-life[1]. Human players while playing against or with computer players, which are used to replace humans, have a few expectations such as predictability and unpredictability, support, surprise, winning, losing and losing well[2].

The goal of agents in *game AI* is similar to the machine used in the Turing test, which humans cannot identify whom they are answering to. [3] organized a game bot programming competition, the BotPrize, in order to find answers to the simple questions such as, can artificial intelligence techniques design bots to credibly simulate a human player?, or simple tweaks and tricks are effective? Competitors submit a bot in order to pass a "Turing Test for Bots". It is relatively easy to identify bots in the system. Some general characteristics used to identify bots were [3]:

- Lack of planning
- Lack of consistency - 'forgets' opponents behavior
- Getting 'stuck'
- Static movement
- Extremely accurate shooting
- Stubbornness

Primary reason for exhibiting such behavior is that these bots are usually modeled using finite-state machines(FSM) and programmed using simple conditional statements, resulting in a very predictable bot to an experienced game player[4]. 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 need to configure many crisp values in their hard-coded logic. Resultant, bots lose their credibility as a human being.

Believability plays a major role for the characters in books and movies, even if it is fiction. Similarly, believability and credibility also plays a major role in video games especially with the artificial characters. However, it is more challenging to design an artificial character for a video game compared to the characters in the books and movies. Characters in video game need to constantly interact with the humans, and adapt their game play without any guidance. There are wide varieties of situation to cope with, and present variety of challenges such as real time, incomplete knowledge, limited resources, and planning[5]. Therefore, the ability to learn will have advantage in increasing the believability of the characters, and should be considered as an important feature[6].

Methods of machine learning could be used effectively 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.

This paper investigates the use of Q-learning, a type of reinforcement learning technique(RL), to improve the behavior of game bots. Q-learning is relatively simple and efficient algorithm, and it can be applied to dynamic online learning. We developed our own game platform for experimentation, a highly simplified simulation of FPS games like Counter-Strike. Bots, which uses learning algorithm, in all our experiments are modeled as terrorist agents. Goals of these agents include killing counter-terrorists, planting the bomb on critical locations, or surviving till the end of the game play.

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. This paper 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:

- *learning to fight*: testing if and how well bots could use RL to learn to fight, and how the resulting performance would compare to human programmed opponent bots,
- *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
- *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.

The rest of the paper is organized as following. Section II provides a background on the FPS game of Counter-Strike and the model of the bots use in such games. Related work in presented in section III. Details of the simulation and methodology are described in section IV. In Section V results of experiments are presented. Conclusion and future work are discussed in section VI.

II. Counter-Strike and Bots

A. Counter-strike

Counter-Strike is a 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. Our initial attempt was to conduct the experiments on a modification of the actual game, but due to improper documentation and complexity of the available source code we developed a scale down simulation of the game. Nevertheless, most of the discussions and the experiments conducted in the paper are inspired from this game.

One of the typical game playing scenarios in Counter-Strike is bomb-planting scenario. There are two teams in the game,

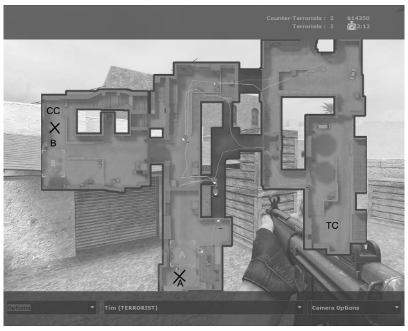

Fig. 1. Game Map

namely terrorist and counter-terrorist. The terrorist aims to plant the bomb while counter-terrorist aims to stop them from planting the bomb. In the processes, sub-goal of each team is to eliminate all the opponents by killing 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 plant the bomb. On the contrary, a counter-terrorist defend these bomb sites and if a bomb gets planted by the terrorists, then counter-terrorists attempt 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.

We simulated very similar game environment, as presented in this section, with an exception of game map; our game map is relatively simple

B. Bots in computer games

Counter-Strike uses the *bots*, also called Non-player Characters(NPCs), to simulate human players in the teams to give the 'illusion' of playing against actual game players. Bots play as a part of the team and achieve goals similar to humans. 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, such as Half-Life, Quake and Unreal-Tournament, with similar methods of programming. 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 starts searching for enemies/opponent. After the opponent is spotted, it makes a transition to attack state in which he fires

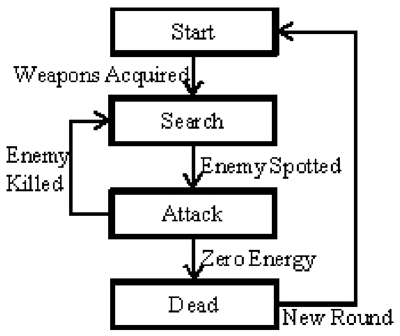

Fig. 2. A PROTOTYPICAL FSM FOR A BOTS

the bullets at the opponent. A Bot may kill an opponent. In that case, it will again start searching for other opponents. Also, a bot could be in any of the above mentioned states and might get killed by the opponent.

There are inherent flaws in using classic FSM model for bots. All transitions/rules needs to be hardcoded in bots logic, and programmers have to spend time configuring these parameters. For example, a rule for agent's attacking behavior based on agent's speed and energy is shown in algorithm 1. Parameters such as energy, distance of enemy, etc., needs to be configured, for which programmers run a large number of time-consuming simulations. Moreover, the use crisp value

Algorithm 1 Hardcoded rules

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

in decision making makes these bots predicable. As a result, it makes a game less interesting and less believable to an experienced game player and eventually may lose interest in the game.

III. Related Work

There are several recent attempts for using machine learning techniques, especially reinforcement learning, for developing a learning bot. Reinforcement Learning (RL) is a machine learning technique where an agent learns to solve problems while interacting with the environment[7].

[8] suggested a learning algorithm to investigate the extent to which RL could be used to learn basic FPS bots behaviors. Their team discovered that using RL over rule-based systems rendered a number of advantages such as: (a) game programmers used minimal code for this particular algorithm, and (b) there was a significant decrease in the time spent for tuning up the parameters. Also, the applied algorithm was used to successfully learn the bots behaviors of navigation and combat, and the results showed that by changing its planning sets of parameters, different bots personality types could be produced. Thus, the paper suggested how an agent can learn to be a bot with the help of RL in shooter games[8].

[9] demonstrates several interesting results using RL algorithm, Sarsa, for training the bots, yet again signifying the effectiveness of such learning techniques. They designed a testbed 2D environment with walls creating partitions on the map. Preliminary experiments were conducted for the bots to learn the Navigation task. Based on the rewards, the bots learn to minimize collisions, maximize distance travel, and maximize number of items collected. After certain number of iterations the bots started receiving greater number of rewards signifying that the bots have learned a positive desired behavior. On manipulating the values of the rewards, the bots learn different behavior. For example, increasing the penalty for collision the bots would learn to remain away from wall, simultaneously ignoring collectible items near the wall. Although, the bots did not met the industry standard their experiment demonstrated that with right parameters the behavior of the bots can be controlled. They also demonstrated the bots learning Combat behavior. Results were similar to navigation task, whereby the rewards for the bots increased after certain number of iterations proving that bots are learning fruitful behavior. Nevertheless, these experiments demonstrate successful use of reinforcement learning to a simple FPS game [9].

Primary reason for using testbest instead of actual game in this this paper and in [8][9] was to reduce c.p.u. cycle and difficulty in dealing with complexities involving in coding for actual game. Nevertheless, several efforts include the use of reinforcement learning in actual video game [10][11]. [10] designed a bot, RL-DOT, for a Unreal Tournament domination game. In RL-DOT, the commander NPC makes team decision, and sends orders to other NPC soldiers. RL-DOT uses Q-learning for making policy decision[10]. There are efforts to develop a NPC that would learn to overtake in racing game like The Open Racing Car Simulator (TORCS). It is suggested that, using Q-learning sophisticated behaviors, such as overtaking on straight stretch or tight bend, can be learned in a dynamically changing game situation[11].

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 [4]. 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. 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.[12].

Widely used AI techniques include RL, neural networks, genetic algorithm, decision tree, Bayesian networks, and flocking [13][14].

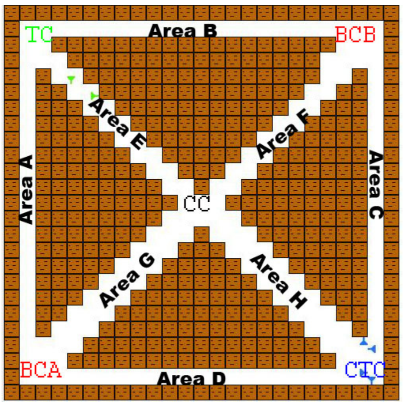

Fig. 3. MAP DIVIDED INTO AREAS

IV. APPROACH

A. Simulation Environment

We developed a scaled down abstraction of Counter-Strike in Java, and simulated the bots in this environment. The miniature version of a Counter-Strike 2D map is shown in figure 3. Herein, bricks are visible which form the boundary of the map and act as obstacles for the 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-terrorists respectively from Counter-Strike. Moreover, there are two sites labeled BCA and BCB in figure 3 which are similar to the bomb sites in Counter-Strike. In figure 3, sites labeled TC and CTC 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' goal is to plant the bomb in one of the sites (either BCA or BCB) or kill all blue agents. Blue agents 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.

B. Methodology

We investigate the Q-learning algorithm for improving the behavior of the *green agents*, while keeping the blue agents' behavior static.

The static Blue agents run a simple algorithm (Algorithm 2), in which if they spots a green agent they shoots a new missile, else they continue moving on map according to *plan*. A plan is a sequence of locations such as TC, BCB, etc. Each agent navigates on the map in the order specified in the plan. Agents randomly choose from six such manually configured plans.

Algorithm 2 Static Blue Agents

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

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 - Q(s,a)***, 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 a 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 : $Q : S \times A \rightarrow R$

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:

$$Q(s_t, a_t) \leftarrow$$
$$Q(s_t, a_t) + \alpha_t \times [r_{t+1} + \gamma \arg\max_a Q(s_{t+1}, a_t) - Q(s_t, a_t)],$$

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

It is necessary to formulate the problem in terms of states and actions for applying q-learning algorithm. Hereby, figure 3 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. The agents will select randomly one of the six manually configured plan. Agents' second state is ***plans***: set $P = \{0, 1, ..., 5\}$ of size six. In addition, the agents use ***enemy present*** of size two as state: set $E = \{0, 1\}$, which signifies whether opponents are present in the individual agents' range(0) or not(1).

Hence, we have state space of 96 states ($A \times P \times E$). An agent will be in one of these states at any period in time, will perform one of the following actions: Attack(0) or Ignore(1).

The game being highly dynamic, it is infeasible to predict the agents' future state and determining whether an action currently executed is fruitful or not (rewards) in order to recalculate utilities. Therefore, the an agent's reward is known in a future state. Hence, utilities of a state s_t is updated when an agent is in a state s_{t+1}. In state s_{t+1} we can determine an agent's rewards for the action attempted in the state s_t and s_{t+1} is treated as the future state for updating the utilities for state s_t. Similarly, 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 explode 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}.

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 greater 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 greater 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: $setPreviousState(currentState)$
```

Algorithms 3 summarize the algorithms used by green agent wherein it is learning the best action i.e. attack or ignore(0 or 1). 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.

V. Experiment

We examine designing agents using Q-learning which can learn different behaviors based on the 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). Therefore, our preliminary experiments are to determine the right combination of parameters.

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.

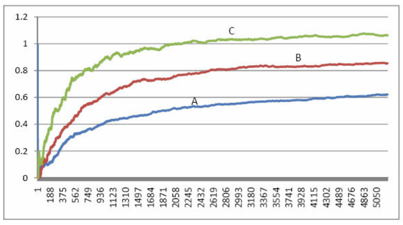

Fig. 4. Learning Curves for Learning to Fight: (A) α = 0.10, γ = 0.90 and, ϵ = 0.1, (B) α = 0.30, γ = 0.95 and, ϵ = 0.1, and (C) α = 0.56, γ = 0.99 and, ϵ = 0.1

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 where one team wins and another loses. Green agents won by killing all the blue agents or planting the bomb. Blue agents won the round by killing all the green agents. For each experiment we modified reward function so that agents can learn differently. The remaining section provide detailed experimental setup and results.

A. *Learning to fight*

For the first experiment, we wanted to train the agents to learn combat behavior. The agents had only two actions to choose from: Attack or Ignore opponents. In the attack action, the 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 the 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.

Figure 4 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 α = 0.56, γ = 0.99 and, ϵ = 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 4. 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 combat behavior.

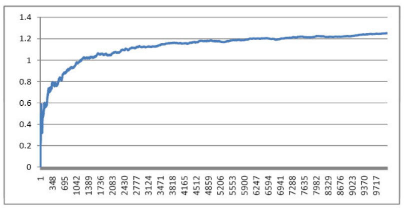

Fig. 5. Learning Curve for Learning to Plant the bomb

B. *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 a better action. Now, we want to explore whether the bots can evolve to learn similar behavior (or better) if they are focused on planting the bomb. Again, the agents had two actions to choose from: attack and ignore. But, now planting the bomb action was part of ignore action. 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.

The agents did not receive any rewards for killing enemy agents. Instead, rewards are given only when an agent plant the bomb: +4 reward for each unit of bomb planted. Similar to the first experiment, the agents are 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 are set to zero before training.

The best fitness ratio is for the combination of α = 0.94, γ = 0.99 and, ϵ = 0.20. Again, the learning rate is high even though the graph shown in figure 5 is quite smooth. The discount rate remains the same. Herein, exploration rate is high due the fact that agents are not receiving any rewards for killing the opponent agents. Yet, in order to successfully plant the bomb an agent has to kill the opponent agents otherwise it will get killed by them. 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 (bomb sites). It also evident from the figure 5 that it is required to run this simulation for more number of rounds.

Table I 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, for example state: (0 H N). 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(not shown in table) because green agent while using plan 4 and 2 never encounters the enemy agent. Therefore, all the enemy present state 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 (0 A n).
- 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 (0 A p).

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. 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.

The bots generated with this experiment outperformed the static bots and learned to attack even though they are not receiving any direct incentives. However, we cannot compare the results of the experiments in the previous section i.e. learning to fight with this experiment. In current experiment, i.e. learning to plant the bomb, definite goal of the agents is to plant the bomb. We modified plans to achieve this behavior such that final location of each plan is one of the bomb sites (BCA or BCB). This action affects the outcome of the game

TABLE I
Q-Table for Plan 0 and 4

State	Attack	Ignore	State	Attack	Ignore
0 A n	14.72	19.33	4 A n	-0.10	0.00
0 A p	2.10	1.83	4 A p	0.00	0.00
0 B n	26.89	35.14	4 B n	-0.10	0.00
0 B p	41.38	32.92	4 B p	-0.10	0.00
0 C n	33.67	33.65	4 C n	-0.10	0.00
0 C p	41.02	33.10	4 C p	-0.10	0.00
0 D n	1.25	1.19	4 D n	0.00	0.00
0 D p	-0.09	1.89	4 D p	0.00	0.00
0 E n	18.54	18.47	4 E n	-0.10	0.00
0 E p	34.33	23.51	4 E p	-0.10	0.00
0 F n	19.58	19.57	4 F n	-0.10	0.00
0 F p	19.99	19.98	4 F p	-0.10	0.00
0 G n	1.47	1.58	4 G n	0.00	0.00
0 G p	2.26	1.82	4 G p	0.00	0.00
0 H n	0.00	0.00	4 H n	0.00	0.00
0 H p	0.00	0.00	4 H p	0.00	0.00

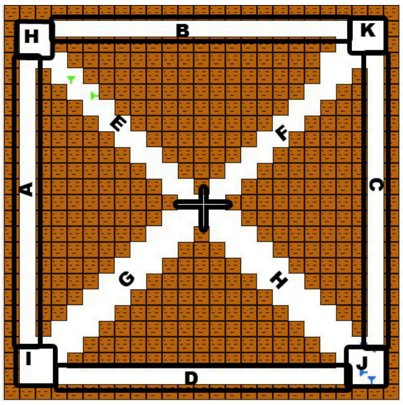

Fig. 6. 12 Areas

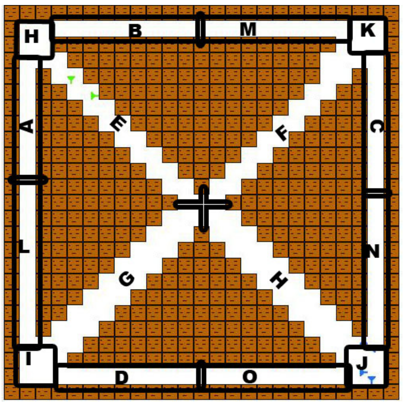

Fig. 7. 16 Areas

because in the former case agents were moving randomly, but now they have a definite goal of going to a particular location and planting the bomb.

C. *Learning for deployment*

Above experiments showed that with current technique the competitive bots can be produced but, the bots with random initial values cannot be supplied with the actual games. So the 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. In this experiment bots are trained for a small number of rounds with the agent having fewer states and then use those Q-values to train the agents with large number of states. Rewards and other settings for the experiment is similar to the experiment is section V.B and used parameter combination α = 0.94, γ =0.99 and, ϵ = 0.20 for experimentation which produced the bots with maximum fitness.

Until now, in all the experiments the map is divided into 8 areas. For subsequent experiments, the map is first divided into 12 areas and then into 16 areas. For a map divided into 8 areas the size of the Q-table is 96 which will increase to 144 for 12 areas and 192 for 16 areas. Figure 6 and Figure 7 shows the divided map for 12 areas and 16 areas respectively. Note that the new divisions are subset of atleast one division from the original map (8 area).

Agents are trained on the map with 8 areas for 500, 1000, 1500, and 2000 rounds and the utilities for the agents are stored. These stored utilities are then used as initial utilities for the agents to be trained on the 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 6 was part of area A in figure 3 therefore, utilities of all the states with area I is set equal to utilities of state with area A. For comparison purpose, we also ran simulation for 12 and 16 areas stating with all zeros in q-table (without fetching initial q-values from 8 areas), and called them the results with 0 initial utility value.

TABLE II
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

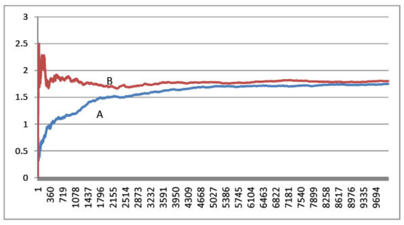

Fig. 8. Learning Curve with 12 areas, where (A) is the curve without any initial training and, (B) is the curve with 1000 initial training from 8 states

Table II shows the highest ratio achieved by agents in each setup, i.e. for 12 and 16 areas. Similar values are observed 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 about this experiment is visible in graphs of figure 8 and figure 9. 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 result shows that initial training provided to the agents with fewer states is useful and the agents exhibit a 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 wins which keeps on fluctuating due to less samples.

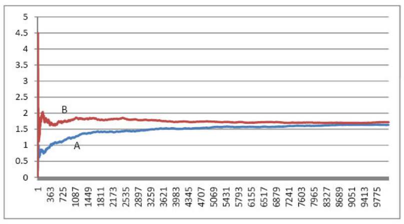

Fig. 9. Learning Curve with 16 areas, where (A) is the curve without any initial training and, (B) is the curve with 500 initial training from 8 states

TABLE III
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

Finally, table III shows comparison between few selected samples from the Q-table before and after training in 12 areas for 20000 rounds. Before training, the utility values of the attack action, when an enemy was present, is less than the value of the ignore action. But after the training an agent's utility value of the attack action is greater than value ignore action. Here, an agent 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 are used as initial value for detailed states, provided a fitness boost to the agents. The agents thereafter remains at the competitive fitness level against the static agents and continue learning a better behavior.

VI. Conclusion

It is evident from the results that the 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 a bot with less number of states we are able to generate a competitive bot for large number of states. In this learning-based approach, the 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 is required for the bots. The bots selected the actions based on its utility values which is 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, the performance of the agents can be improved by devising a method through which agents can learn to select the plan. Currently, a plan is selected randomly after each round; instead, a plan based can be selected based on the past experience of the bots. The 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, a confluence of various learning technique can be used to improve the learning speed of agents. For example, after each round we can use genetic algorithm to mutate utility values in Q-table among the agents in order to generate a better population. Currently, all the five agents are learning separately without any integration 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, the bots need to play against the human players. Although the dynamic bots were tested against static bots, yet human behavior is very unpredictable.

References

[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, pp. 16-20.*

[3] P. Hingston. A new design for a Turing Test for Bots. *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on , 2010, pp. 345-350.*

[4] 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, pp. 139-145 Vol. 1.*

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

[6] F. Tenc, C. Buche, P. D. Loor, and O. Marc. The Challenge of Believability in Video Games: Definitions, Agents Models and Imitation Learning. *GAMEON-ASIA'2010, France, 2010, pp. 38-45..*

[7] R. Sutton and A. Barto. Reinforcement Learning:An Introduction. *The MIT Press Cambridge, Massachusetts London, England, 1998.*

[8] M. McPartland, and M. Gallagher. Learning to be a Bot: Reinforcement learning in shooter games. *4th Artificial Intelligence for Interactive Digital Entertainment Conference, Stanford, California, 2008, pp. 78-83.*

[9] M. McPartland, and M. Gallagher. Reinforcement Learning in First Person Shooter Games. *IEEE Transactions on Computational Intelligence and AI in Games, 2011, Vol. 3.1., pp 43-56. .*

[10] H. Wang, Y. Gao, and X. Chen. RL-DOT: A Reinforcement Learning NPC Team for Playing Domination Games. *IEEE Transactions on Computational Intelligence and AI in Games, 2010, Vol. 2.1, pp. 17-26..*

[11] D. Loiacono, A. Prete, P. Lanzi, and L. Cardamone. Learning to overtake in TORCS using simple reinforcement learning. *2010 IEEE Congress on Evolutionary Computation (CEC), 2010, pp. 1-8*

[12] S. Zanetti and A. Rhalibi. Machine Learning Techniques for FPS in Q3. *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology, 2004, pp. 239-244.*

[13] D. Johnson and J. Wiles. Computer Game with Intelligence. *Australian Journal of Intelligent Information Processing Systems, 7, 2001, pp. 61-68.*

[14] 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, pp. 124-131.*