`

HQ-DoG: HIERARCHICAL Q-LEARNING IN DOMINATION GAMES

by

ALLEN TAYLOR III

(Under the Direction of Walter D. Potter)

ABSTRACT

This thesis presents HQ-DoG, an algorithm that uses hierarchical Q-learning to learn a policy that controls a team of ten soldiers in a video game environment to compete in the gametype known as domination. Domination is a game where two opposing teams compete for possession of three different positions on a map. The team that holds the most positions for the longest amount of time wins the game. HQ-DoG uses three Q-tables that represent a captain and two subordinate lieutenants connected in a hierarchical structure. Together the captain and lieutenants learn where to deploy soldiers across the environment in different circumstances and how many soldiers should be sent to different targets. HQ-DoG is tested against a series of strategies that exhibit different levels of sophistication in playing domination, and the results show it is able to learn a competitive strategy given enough time.

INDEX WORDS: Reinforcement Learning, Hierarchical Reinforcement Learning, Q-Learning, Artificial Intelligence, Games, Video Games, Domination

HQ-DoG: HIERARCHICAL Q-LEARNING IN DOMINATION GAMES

by

ALLEN TAYLOR III

BS, North Carolina Central University, 2008

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF ARTIFICIAL INTELLIGENCE

ATHENS, GEORGIA

2012

HQ-DoG: HIERARCHICAL Q-LEARNING IN DOMINATION GAMES

by

ALLEN TAYLOR III

Major Professor:	Walter D. Potter
Committee:	Prashant J. Doshi Khaled Rasheed

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2012

This thesis is dedicated to my family and friends who helped me and gave me support.

ACKNOWLEDGEMENTS

I would like to thank the faculty of the University Of Georgia Institute Of Artificial Intelligence and the Computer Science Department for everything they have taught me.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS v

LIST OF TABLES viii

LIST OF FIGURES ix

INTRODUCTION 1

Chapter 1 BACKGROUND 3

1.1 Motivation 3

1.2 Domination Gametype 3

1.3 AI and the Gaming Industry 4

Chapter 2 THEORY 7

2.1 Agent Organization 7

2.2 Control Systems 13

Chapter 3 RELATED WORKS 22

3.1 Game Industry FPS AI 22

3.2 Control of Individual Soldiers 27

3.3 Control of a Team of Soldiers 29

Chapter 4 HQ-DoG METHODOLOGY AND IMPLEMENTATION 36

4.1 Simulation Environment 36

4.2 Captain's Process 39

4.3 Lieutenant’s Process 42

4.4 Soldier Deployment 46

4.5 Different Soldier Behavior........ 47

Chapter 5 RESULTS AND DISCUSSION 49

5.1 Opponent Strategies........ 49

5.2. Experimental Results 53

5.3 Discussion........ 69

CONCLUSION........ 71

REFERENCES 73

Appendix A – Captain States and Actions........ 77

Appendix B – Lieutenant State and Actions........ 80

LIST OF TABLES

Page

Table 1 - Captain Q-table vs. (5,5,0). Lost point A & B. Captured point C 61

Table 2 - Captain Q-table vs. (5,5,0). All points neutral.. 62

Table 3 - Lieutenant's Q-table vs. Static Strategy (5,5,0) 62

Table 4 - Captain Q-table vs. (4,4,2). Lost point A. Captured points B & C.. 63

Table 5 - Captain Q-table vs. (4,4,2). Lost points A & B. Captured B 64

Table 6 - Captain Q-table vs. (3,4,3). All points neutral... 65

Table 7 - Captain Q-table vs. (3,4,3). Lost point A. Captured point B. Point C is neutral 65

Table 8 - Captain Q-table vs. (3,4,3). All points captured 66

Table 9 - Defense Lt.'s Q-table vs. Static Strategy (3,4,3).. 66

Table 10 - HQ-DoG vs. Static Strategies (6,2,2) and (8,1,1) 66

Table 11 - HQ-DoG vs. Static Strategy (4,4,2) 1000 Games... 68

Table 12 - HQ-DoG vs. Static Strategy (4,4,2) Second 500 Games 68

LIST OF FIGURES

Page

Figure 1 - Hierarchical organization with managerial and operational roles 9

Figure 2 - Same agents but in different hierarchical organization 11

Figure 3 - Comparison among MDPs, SMDPs, and Options 13

Figure 4 - Task graph of two robots that must cooperate to clean an office room 19

Figure 5 - Evaluation of different moves on chess board. Consider threats from other pieces 23

Figure 6 - Path finding evaluates positions' danger from threat, not just distance to goal 23

Figure 7 - Evaluation of different positions in game. Consider enemy lines of fire and mobility.............. 24

Figure 8 - Three state finite state machine used in F.E.A.R. 25

Figure 9 - Specify the goals (rectangles) and their actions (circles) independently of each other.............. 26

Figure 10 - Automating interaction between goals and actions 26

Figure 11 - A simplified environment used to model Counter Strike 27

Figure 12 - A more complex environment used to model Counter Strike 28

Figure 13 - Task decomposition of HTN 30

Figure 14 - The domination map used for HQ-DoG 38

Figure 15 - Steps of the HQ-DoG algorithm starting from sensing the game environment 39

Figure 16 - Lieutenant Soldier Allotment vs. Captain Target Decision 43

Figure 17 - Different subset of actions A_i can be chosen based on current state 44

Figure 18 - Win/Loss chart of HQ-DoG vs. Greedy Strategy 55

Figure 19 - Score Difference of HQ-DoG vs. Greedy Strategy 55

Figure 20 - Win/Loss chart of HQ-DoG vs. Static Strategy (5,5,0) 56

Figure 21 - Score Difference of HQ-DoG vs. Static Strategy (5,5,0) 56

Figure 22 - Win/Loss chart of HQ-DoG vs. Static Strategy (4,4,2) 57

Figure 23 - Score Difference of HQ-DoG vs. Static Strategy (4,4,2) 57

Figure 24 - Win/Loss chart of HQ-DoG vs. Static Strategy (3,4,3) 58

Figure 25 - Score Difference of HQ-DoG vs. Static Strategy (3,4,3) 58

Figure 26 - Win/Loss chart of HQ-DoG vs. Random Strategy 59

Figure 27 - Score Difference of HQ-DoG vs. Random Strategy 59

INTRODUCTION

This thesis presents HQ-DoG (Hierarchical Q-Learning in Domination Games), an algorithm which controls a team of agents in a video game environment, and attempts to learn a winning policy for the gametype known as domination (see Section 1.2). In domination, there are multiple spots in the game map called domination points that a team must capture in order to score points. The more spots captured, the faster points are accumulated. The challenge is to create an algorithm that effectively directs the team towards a winning strategy. The algorithm has to observe the environment and make decisions about when to capture and defend different spots. This is an important topic in the gaming industry, especially for the first person shooter (FPS) genre where domination is most often used. Players do not always want to compete against other humans; sometimes they are not able to play against humans, perhaps from lack of an internet connection. Other times there is a learning curve to the game that players must overcome to be competitive against other people. These obstacles should not prevent them from having an experience where they can play on a team against another team. That is why games have modes where a person can play with and against in-game agents, called soldiers. It is important that the soldiers seem competent to make the user experience more realistic and entertaining. This includes appearing to work together as a team.

The organization of the team of soldiers can be made very similar to that of a military. There is a person in charge at the top who has the final say on major matters. Under the highest leader there are people with less authority relative to the leader, but who still have enough power to do their separate jobs. The composition of this authoritative body depends on the scale under consideration. At the largest scale, it might be the President and the Joint Chiefs of Staffs, but at the other end of the spectrum, it might be the captain of a platoon and its lieutenants. This smaller side is the scale at which video games use a team of soldiers, and it is the inspiration for the control system used in this thesis. Soldiers are treated as if they are in a platoon. There is a hierarchical structure to the platoon where a captain soldier is the direct superior to two lieutenant soldiers, one in charge of offense and the other in charge of defense. Each lieutenant has a team of soldiers under their command, and together with the captain they direct the entire platoon on what domination points to attack and defend and how many soldiers to commit to each target.

Layered on top of this hierarchical structure is a decision-making process that uses hierarchical reinforcement learning (HRL). Reinforcement learning is a machine learning (ML) approach that teaches agents to make decisions in an environment based on the rewards given for different actions. HRL teaches in the same way but breaks up the decision making responsibility into a hierarchy of tasks and subtasks. Because the soldiers are already modeled as a hierarchy, HRL intuitively maps onto the team. The captain and lieutenants each get their own task to learn. The captain learns to make broad strategic decisions based on the environment and the lieutenants learn the most rewarding way to complete their separate tasks and follow orders given to them by the captain. Soldiers do not learn; they only do what they're told. HRL allows for learning how to control the team during the game, but does not place the entire responsibility on one soldier. It divides the duty as you would see in the army. For these reasons this thesis attempts to show that the combination of hierarchical organization and hierarchical decision-making can produce a winning policy for a team of agents in a militaristic video game environment.

Chapter 1 BACKGROUND

1.1 Motivation

Modern video games use simple artificial intelligence (AI) techniques for tasks such as path finding and simple decision making. As attention shifts from graphical fidelity to realistic behavior and gameplay, more advanced AI techniques will have to be implemented. One of the techniques gaining notice is the use of agents to control non-player characters (NPCs). This thesis employs a hierarchical decision making process which explicitly establishes coordination as well as lines of communication among the NPC agents. It reduces the state-action space of the decision making domain by breaking the space into smaller coherent pieces and assigning those pieces to individual agents at different levels. This divide and conquer strategy allows the overall goal to be achieved by breaking it into sub-goals that can each be accomplished separately. The strategy is applied to a set of agents working together as a team to play the gametype known as domination. In this gametype, the team tries to capture and maintain ownership of three territories in a computer game against another team of agents trying to do the same thing.

1.2 Domination Gametype

Domination is a gametype that rewards overall team strategy and not individual performance. It is defined by key locations on the map called domination points owned by each side. A team captures a domination point by going to the spot and holding it for a specific interval of time after which point it becomes theirs. Players are able to fight each other in the map. When a player dies, they are respawned randomly elsewhere on the map at what are called respawn points. When playing, the advantage doesn’t come directly from killing the opponent. It comes indirectly from capitalizing on the absence of the opponent as a result of their death, giving the player the opportunity to capture a point or continue defending it from future assailants. Good domination gameplay needs coordinated team strategy. Human players don’t control each other while playing, but the best performance occurs when they communicate with each other and work together. Domination games are stochastic, adversarial, and deal with a partially observable environment. Typically a player only knows the position of the enemies that are seen, and perhaps whether or not different domination points are being captured or lost. This is why domination is a suitable test bed for team-based AI (Hogg, Lee-Urban, Munoz-Avila, Auslander, & Smith, 2011).

1.3 AI and the Gaming Industry

Video games are becoming ever more popular and influential in the entertainment industry both domestically and abroad. A recent video game from game publisher Infinity Ward called Call of Duty 4: Modern Warfare 2, released in November of 2009, broke worldwide records for its initial 5 day sales figures. In its first week it sold over 5 million copies (Walton, 2009) and had total overall sales of over $550 million. At the time this was more than any other form of entertainment in history including movies, music, and books (O'Connor, 2009). Two years later the sequel to this game (Call of Duty 4: Modern Warfare 3) had even bigger sales. It reportedly sold 6.5 million copies and $400 million, in its first 24 hours (MacDonald, 2011).

As the hardware on which games are played pushes the limit of how realistic the graphics can be, more attention will be paid to the actual game play. Focus will change from graphical fidelity to more complex behavior including natural interaction among NPCs and higher cognition used to control them (Dignum, Westra, van Douesburg, & Harbers, 2009). In the future, in-game AI will be an increasingly important factor in making games more interesting, challenging, and fun. The majority of AI in contemporary games is controlled by scripting, finite state machines (FSMs), some sort of rule-based system, or behavior trees (Miikkulainien, 2006), (Bourg & Seeman, 2004), (Champard, 2007). These techniques are limited though because they fall back on the designer to anticipate different scenarios the player might encounter in the game. Unfortunately even the most impressive AI done in this fashion can become repetitive and then the focus of the game is reduced to memorizing the patterns and defeating them (Miikkulainien, 2006). Future games must contain AI that is capable of more than just immediate reactionary behavior. They must be capable of making plans then acting to achieve those plans. This cannot be done with state-machines (Dignum, Westra, van Douesburg, & Harbers, 2009).

FSMs have the advantage of providing a simple mechanism to edit behavior, but they don't scale well to large systems. Also their code isn't modular; states are not reusable for different goals without connecting them in a different way. Hierarchical FSMs (HFSMs) are used to solve the issue of scalability, but they still don't solve the problem of modularity and reusability. Behavior trees are the latest popular tool to control game behavior, and they try to ensure modularity and reusability by removing the state-transition logic that would normally be found in an FSM. States become self-contained behaviors with a collection of actions without transition code to an external state (Champard, 2007). Behavior trees have been used in recent games like Halo Reach and Red Dead Redemption. They have been applied to areas like squad logic and strategy AI. Since 2011 a second generation of behavior trees has been employed in commercial games, and they have more layers of decision that build up and more options to choose from than their smaller first generation counterparts (Champard, 2012b). However, behavior trees are still static and rely on the domain knowledge of their designer to create proper behavior.

The future of the game development industry lies with advanced AI technologies like multi-agent systems (MAS) and ML, but despite the benefits of advanced AI, retail game producers have been somewhat reluctant to take advantage of them over the past few years. There are numerous possible reasons for this. One may be that ML and MAS are designed to adapt and are thus non-deterministic by nature. This makes it extremely difficult for developers to test and debug (Bourg & Seeman, 2004). Non-deterministic behavior may also show up in the game itself, even after testing, as a result of the learning process, which could produce undesirable game play (Mikkulanien, Bryant, Cornelius, Karpov, Stanley, & Yong, 2006). It's the responsibility of the game developer to create an entertaining experience with agents that seem intelligent from a human's point of view. Game developers have to satisfy the subjective notion of what it means for a game to be "fun", a constraint researchers may not necessarily have to contend with. Researchers must build algorithms that prove the efficacy of their methodology. While they do have to deal with issues like efficiency and practicality, researchers' primary concerns are liable to be more academic than anything else. "Fun" is not so much an issue as "effective" is. These two obligations are not always reconcilable (Mikkulanien, Bryant, Cornelius, Karpov, Stanley, & Yong, 2006).

Another possibility is that it may just be the fact that the present level of technology within entertainment software has proven adequate to drive games to their current degree of success. 2011 was a year of many sequels including Gears of War 3, Call of Duty 4: Modern Warfare 3, Uncharted 3, Killzone 3, Crysis 2, and F.E.A.R. 3, which some in the game industry have attributed to a lack of innovation (Champard, 2012a) Game producers might feel incorporating advanced AI into games is too big a risk to invest in right now (Miikkulainien, 2006). Due partly to the recession, research budgets have been reduced, and games have developed as much of the simple AI as they can so progress into more advanced techniques has been slow. On top of that it has been a problem for AI developers to convince game designers that AI can be incorporated into areas such as animation, behavior, and reasoning to make the entire game programming endeavor faster and cheaper with less effort (Champard, 2012a). In the end it may simply fall on researchers to prove the efficacy of advanced AI techniques if there is to be any significant move on this front.

From the perspective of the AI community, games form a good test bed for ML and MAS simulation. Many research issues can be safely applied and studied in games like real-time reactions in a stochastic environment, management of multiple (possibly conflicting) goals, team work, and the incorporation of personality and emotion into agents as well as their effect on behavior (Lees, Logan, & Theodoropoulos, 2006). Early AI for games used symbolic knowledge representation because it worked well with card games and board games like chess, checkers, backgammon, and poker. Modern video games have highly dynamic environments with multiple characters both player-controlled and NPCs. FPS games like the F.E.A.R, Killzone, and Unreal Tournament franchises are good examples of this. Their NPCs must respond quickly to ever fluctuating environments that have things like the collapse of buildings and infrastructure and the changing of objective positions like in capture the flag. ML techniques thrive in the type of environment found in modern games. As such, these types of games should be considered a viable platform to test and commercialize more advanced ML and multi-agent techniques (Miikkulainien, 2006).

Chapter 2 THEORY

2.1 Agent Organization

One of the most fundamental aspects of designing a MAS is deciding how the agents are to be organized within it. This is a very important question because its answer will dictate other critical factors like the amount of agents the system can handle, the degree of cooperation among the agents, what type of communication is allowed, if any, and the protocols needed to facilitate it all (Kraus, 1997). Organization deals with how the agents are structured and controlled and that structure can be either centralized or decentralized depending on the needs of the developer and the nature of the environment in which the agents will be deployed. Both have pros and cons that must be studied carefully.

2.1.1 Decentralize vs. Centralized

Decentralized control is often employed on systems that require a distributed architecture, over a network for example. Agents in this type of system usually have a greater level of autonomy because they have more decision making responsibility. There is no higher authority telling them what to do, so they must make many choices for themselves (Dignum, Dignum, & Sonenberg, 2004). An example of this is in team sports, like Robo Soccer, where agents have to individually decide on their own actions (Mohd Shukri & Mohd Shaukhi, 2008). Web programs that “crawl” the internet are another example of decentralized control. Agents might go out searching through webpages then return with the desired information without a centralized hub directing where they go or how they behave as they search. It is common to approach MAS as a gathering of autonomous agents with little structure. The resulting system is dynamic in that agents are not burdened with a centralized thinking process that may not work well in a distributed environment. On the other hand, it becomes more difficult to collectively control the agents. Communication becomes harder to implement, and if there is no explicit communication, any visible cooperation is potentially emergent and not necessarily designed (Ferber & Gutknecht, 1998).

More centralized structure is needed for the purposes of this thesis; the goal is to coordinate the agents in a team effort to achieve multiple objectives. A chain of command where supervisor agents hand down instructions to their subordinates is used to do the job instead of relying on fully autonomous agents and emergent behavior. There are three levels in the chain. A captain is at the top, in charge of the entire team. At the second level there are two lieutenants, one responsible for offense, the other responsible for defense. At the bottom of the chain of command are the soldiers that complete orders given to them. Neither the captain nor the lieutenants are actually in the game environment; they are only represented abstractly by the orders they give. Only the soldiers are embodied with a virtual presence inside the environment. Although not as flexible as a decentralized approach, this allows the captain and lieutenants to evaluate the environment and make global decisions for the entire team. An explicit illustration of global constraints, represented by the state of the environment, and the solution, represented by the actions being executed, can be kept at all times (Bensaid & Mathieu, 1997). It is much harder to do this with a decentralized system where each agent has its own view about the world state. Not all the agents require higher decision making abilities; the soldiers need just enough to carry out the low-level orders they are given. This compartmentalization of responsibility allows each agent to focus and limit its reasoning to the best action for its immediate responsibility (Dignum, Dignum, & Sonenberg, 2004).

This sort of team effort with different objectives lends itself to a hierarchical organization of the soldiers, as in the military or a large corporation (Wooldridge, Jennings, & Kinny, 1999). Hierarchies are natural structures for task delegation among agents (Routier, Mathieu, & Secq, 2001) because they are able to break down complex tasks. An agent decomposes a problem into smaller problems and assigns them to agents lower down in the chain. Agents repeat this process until problems are decomposed to the point where they can be solved by single agents. Divide and conquer allows the hierarchical approach to scale well with larger organizations of agents. It also clearly delineates authority and responsibility among the agents by giving each agent a role (Ghijsen, Jansweijer, & Wielinga, 2010).

2.1.2 Agent Roles and Relationships

MAS organizations are defined by both their structure and the roles their agents play. This includes the way the agents are arranged, the way they interact, and the relationships among their respective roles (Ferber & Gutknecht, 1998). These roles are characterized by the set of skills needed to fulfill them. Skills are the competencies an agent has and they govern the different roles an agent can play in the organization (Glaser & Morginot, 1997). When these competencies are implemented as a set of coherent functions it allows roles to be dynamically assigned to and withdrawn from agents as the situation demands. This promotes modularity and reuse of functionality (Routier, Mathieu, & Secq, 2001).

The relationship among roles is directly dependent on the hierarchical structure of the organization. The two most basic roles are operational and management. The hierarchy allows a manager agent to be in charge of multiple agents at a lower level, irrespective of their roles. Any agent that is the boss of another agent must at least play a manager role. As long as the rules permit, an agent can play both a manager and operational role at the same time. An operational agent performs tasks that impact the completion of the organization's goals. Manager agents are responsible for the coordination of these operational tasks (Ghijsen, Jansweijer, & Wielinga, 2010). By enforcing these rules, a tree-like structure is formed among the agents, as in Figure 1. To monitor goal progress and implement actions, information must pass among the agents.

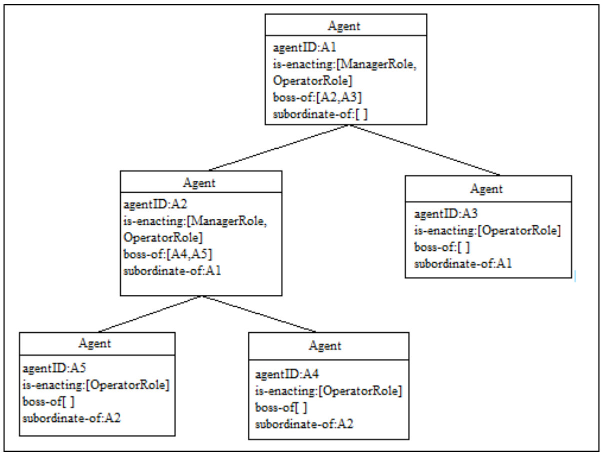

Figure 1 - Hierarchical organization with managerial and operational roles

In a hierarchy, information flows down the structure in the form of orders and requests for new information from upper management. Lower level agents use this information to carry out their orders or gather intelligence for the requests. Information flows back up the architecture pertaining to the success or failure of objectives and other data pertinent to maintaining and updating plans. This type of architecture allows higher levels to pass down goals and context while the lower levels send up sensory information (Atkin, Westbrook, & Cohen, 1999). The information that travels up the chain can come from multiple sources simultaneously and converge into a single manager. This information might need to be aggregated, meaning the information is fused together, abstracting it and making it simpler to understand. Abstraction makes it easier for boss agents to receive potentially large amounts of information from multiple subordinate agents. An agent in a management role needs to know the capabilities, status, and responsibilities of the agents under its control (Ghijsen, Jansweijer, & Wielinga, 2010).

2.1.3 Organizational Reconfiguration and Adaptation

Because organizational structure and agent roles are related, a change in one may change the other. Structural change is defined as moving an agent to a new position in the hierarchy somewhere other than its current place. The general hierarchical nature is maintained, but its tree-like structure before and after the reconfiguration is not identical. One of the main reasons for allowing the structure to be able to change is so that it can adapt to a stochastic environment. For example, the lieutenant in charge of the team's offense may need more soldiers under its command, meaning soldiers will have to be taken from under the defensive lieutenant's chain of command. With respect to the hierarchical representation of the organization, this corresponds to shifting the selected soldiers from the branch of the tree under the defensive lieutenant to another branch under the offensive lieutenant. Adaptation of this type requires that the system be able to monitor its current status and implement actions to preserve or salvage it. These actions include reconfiguration of agent positions in the structure as described above.

When it comes to roles, the distribution of roles to the agents in the organization can be thought of as the current behavioral convention that the system employs. Dynamic adaptation in terms of the relationships among agents means changing from one convention to another with the reassignment of roles (Glaser & Morginot, 1997). Take the structural change described above for example. The soldier agents under the defensive lieutenant all have defensive roles which may be drastically different from the roles played by the soldiers under the offensive lieutenant. If the offensive lieutenant receives more resources, and solders are transferred under its command, this constitutes a new behavioral convention because there has been a redistribution of roles. The agents that once played a defensive role are now playing an offensive role which, depending on the implementation of the system, may accomplish a completely different task. The structural reconfiguration implicitly causes a reconfiguration in roles which can lead to a change in behavior for the entire system, namely one that exhibits more aggression. The same principle holds true when switching agents from an offensive to defensive role. Figure 2 has the same agents that are presented it Figure 1. It demonstrates how agents in one hierarchical structure can be rearranged into another with a different role assignment.

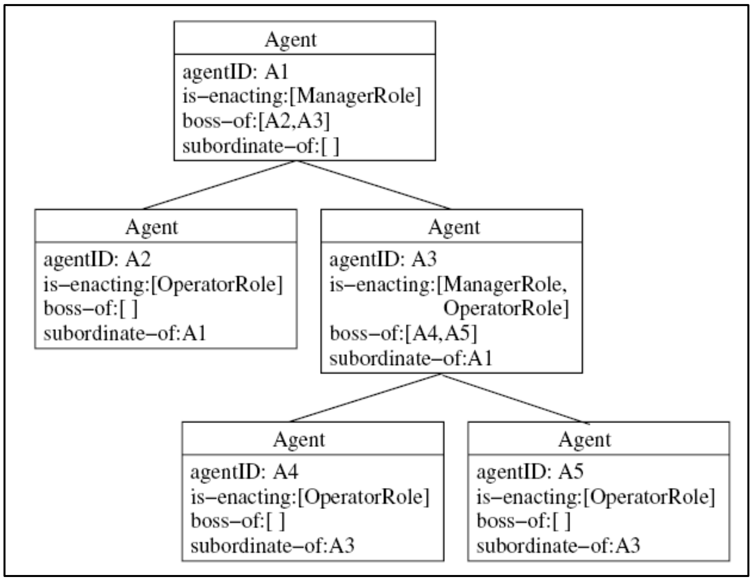

Figure 2 - Same agents but in different hierarchical organization

Reconfiguration should increase the overall utility of the organization, meaning the team somehow performs better in the new arrangement than it did in the previous one. Managerial agents in charge of reorganization, in this case the captain and lieutenants of the team, must reason about changes in the environment and decide how the system adapts to those changes (Dignum, Dignum, & Sonenberg, 2004). The alternative to developing a system that can adapt is trying to build a system robust enough to predict and handle all foreseeable events. However this is difficult to accomplish as it is nearly impossible for a designer to predict all the conditions of a stochastic environment. Therefore it is better to give the organization a degree of flexibility; the approach of dynamic adaptation typically performs better than prediction (Ghijsen, Jansweijer, & Wielinga, 2010).

By changing its structure and roles, a dynamic architecture is able to balance its workload among the agents and ensure that no bottlenecks occur (Routier, Mathieu, & Secq, 2001). However, the change must be orchestrated by an agent in a managerial role, which can happen in various ways. A manager agent can tell a subordinate agent to abandon or adopt a specific role, be it operational or managerial. A manager agent can tell a subordinate agent to go work for another subordinate agent. With this change, authority relationships must be broken and recreated somewhere else in the hierarchy. Agents can be thought of as scarce resources of the system, and arbitration of the utilization of those resources is left up to managerial agents (Atkin, Westbrook, & Cohen, 1999). This allows agents to be assigned different tasks as the need arises (Ghijsen, Jansweijer, & Wielinga, 2010). For example in a domination game, more agents may need to be assigned to help capture a specific point if the enemy is putting up too staunch a resistance. On the other hand more agents may need to be assigned to play defense if there is a comfortable lead over the opposing team and it is more prudent to try to defend the currently possessed points rather than spread the team thin trying to capture more points.

The problem with doing rearrangement is finding the right distribution of agents across the hierarchy. Just because an agent can be moved or given a different role does not mean that it should be. In the case of this thesis, there is the danger of becoming too offensive or too defensive at the wrong time. Overstretching the forces to capture more domination points can lead to all your forces being defeated and losing all points. However, too conservative an application of forces leads to not enough points gained. Also the process of reconfiguration should not be cumbersome on the team. There is the risk that reorganization can take too long, which could be harmful to the performance of the team in a fast-paced environment like a video game where the opponent team is always active. Reassignment should be efficient and not detract significantly from the execution of other actions.

2.2 Control Systems

Once an organization has been established for the agents, a control mechanism must be put into place. The mechanism is responsible for guiding the actions of the agents and controlling how they respond to changes in the environment, such as when a domination point is lost or when a substantial portion of the team is killed and respawned somewhere else. Traditionally planning is used for action selection; however, planning focuses on search, optimization, and much deliberation before decisions are made. It excels when applied to mathematics or board games where there is time to look for the optimal solution between moves. Dynamic games need a more efficient approach because they are typically fast-paced and real-time, thus the use of a control system would make better sense. Control systems do not have the luxury of extensive online planning. Online planning searches for the best solution to a problem concurrently as the algorithm is running as opposed to offline planning, which searches before run-time and then implements the solution when needed. Action is continuous and time sensitive because lag between observation and action can lead to destabilization and poor performance (Albus, Anthony, & Roger, 1981). Given these constraints it is not the absolute goal of a control system to find the optimal solution, but to find the best possible solution within a reasonable amount of time. There are many approaches to accomplish that.

2.2.1 Markov Decision Processes

A common methodology for agent decision making is the use of Markov Decision Processes (MDPs). MDPs choose their next course of action based solely on the current condition of the environment. Previous states and choices are not taken into account. They can be used for games because games typically have large state spaces, need problem solving to succeed, and often use sequential decision making (van Eck & Wezel, 2008). MDPs are characterized by the tuple $\langle S, A, T, R \rangle$ which define their states, available actions, transition probabilities, and immediate rewards. With these a policy and performance metric are defined as well. State is defined by the one or more variables required to describe the system at any given time. Actions are the controls or commands issued by the system, dependent on the system’s current state. Actions directly affect the next state of the system. A state-action pair is the tuple of a state of the environment and one of the possible actions that can be taken from that state. Associated with each state-action pair is a probability that the system will successfully transfer from the current to desired state given the action taken. These probabilities grow exponentially with the number of states and actions. After each transition from one state to another, the system receives an immediate reward, either positive or negative, indicating the utility of being in that state. A policy is the directive that specifies the course of action to take from each possible state. A performance metric grades the overall quality of a given policy. The general point of the MDP is to choose the optimal policy that maximizes the utility of the system (Gosavi, 2009).

2.2.2 Dynamic Programming

MDPs are traditionally solved with a class of algorithms that are guaranteed to produce the optimal policy, known as dynamic programming (DP). DP is a recursive technique that solves a small subset of the MDP then iteratively finds the optimal policy for larger areas of the state space. The problem with DP is that it is susceptible to poor performance in the presence of two conditions. The first condition is called the curse of dimensionality and occurs when the state space of the domain is too large, in the millions for example, and it becomes implausible to try to store all the states. The second condition is called the curse of modeling. DP requires perfect knowledge of the domain in terms of the random parameters of the system and the exact values of transition properties from one state to another, and it can be difficult to derive these based on the nature of the model (Gosavi, 2009) (van Eck & Wezel, 2008). Reinforcement learning (RL) is a technique used to solve MDPs similar to DP but does not suffer from the same problems. The two are similar in that they both seek to learn value functions in order to find an optimal policy, however RL does not have to cope with the curse of modeling. It learns value functions by interacting with the environment directly or a simulation thereof (Gosavi, 2009). Instead of using transition probabilities, it repeatedly observes the consequences of randomly choosing an action from a particular state and records the results to find a near optimal policy (van Eck & Wezel, 2008).

2.2.3 Q-Learning

One of the most common methods for RL is called Q-learning. Q-learning involves storing Q-factors, each of which is comprised of a unique state-action pair of the domain. Associated with each Q-factor is a real number which is initially set to a sufficiently low number (usually zero). Learning commences by running a simulation of the domain in which the learner repeatedly chooses actions based on the current state and then receives feedback based on the utility of the following state. If the feedback for arriving at a certain state based on a previous action is good then the Q-factor of that previous state-action pair is increased; if the feedback is negative the value is decreased. After a large number of steps have been taken, all the Q-factors associated with each state are evaluated. The factor with the highest value is deemed the best, and thus the action tied to that factor will be the one chosen for that state in the future (Gosavi, 2009). Algorithm 1 shows the Q-learning algorithm.

Algorithm 1 Q-Learning Algorithm

```
1. Initialize each state-action pair (s,a), $Q(s,a) \leftarrow 0.0$
2. Observe State
3. Choose action
     • $a \leftarrow \arg max_k\, Q(s,k)\ with\ probability\ 1-\ \varepsilon$
     • $a \leftarrow random\ action\ with\ probability\ \varepsilon$
4. Perform action and observe new state $s'$. Calculate reward r.
5. Update Q-table for previous state-action pair
     • $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma\ max_{a'}\, Q(s',a'))$
6. Current state becomes new state. Go to step 3
```

In the above algorithm, α is the learning rate and it controls how much influence the utility of the current state and the max expected utility from this state will have on the Q-factor of the previous state. The discount factor is represented by γ and determines how important future rewards are to the decision process, whether the learner values immediate reward over long-term gain. Long-term gain is calculated by finding the Q-factor whose action maximizes value over all possible actions that can be executed from the current state.

The typical way to do Q-learning is to save the value of each Q-factor in what is called the lookup table approach (using a Q-table). However, Q-learning does not escape the curse of dimensionality. It cannot use a lookup table if the state-action space is too large. An alternative is to use a function approximation technique like a neural network or linear regression to map the state-action pairs to Q-values. A neural network can train on a subset of the state-action space and then is able to generalize and approximate the Q-values for the rest of the space based on key learned features. Neural network Q-learning solves larger problems than the table lookup version, but is not guaranteed to converge to all the correct Q-values for the domain because a change of a single neural network weight does not correspond to the localized change of a Q-value in a look up table. It corresponds to the change of many Q-values at once, some of which may have been the correct one. In spite of this, this form of Q-learning has been proven to be effective for domains with large state spaces, like the game of Othello (van Eck & Wezel, 2008).

2.2.4 Semi-Markov Decision Processes

The problem with regular MDPs is that decisions usually have implications that lead far into the future; there are more than just the immediate consequences to take into consideration. Actions also usually take an extended amount of time to complete, but conventional MDPs only move one time step ahead after each action and then decide the next action. This is where Semi-Markov decision processes (SMDPs) come into play. With SMDPs the time to transfer from the current state to another can be greater than one unit (Gosavi, 2009). SMDPs allow actions to be temporally abstracted and continuous, unlike regular MDPs, whose actions are discrete with time steps that only last 1 unit. Further down in the theory, SMDPs also support the notion of an option. To discuss options, it is necessary to build the distinction between primitive actions and extended actions. Primitive actions are simple and singular while extended actions are more complex and are made up of smaller actions. Standard SMDP theory treats extended actions as regular actions that just take longer to complete than primitive actions. They are like little black boxes whose interior cannot be analyzed. An option, on the other hand, overlays SMDP continuous extended actions onto discrete time MDP actions and analyzes the extended actions in terms of the base MDP simpler actions. It opens up an extended action to be studied in terms of the primitive actions it is made of. With this internal examination comes the possibility to change the policies of the underlying MDP as necessary. Options can thus be considered as plans (not to be confused with planning) that can be changed as needed (Sutton, Precup, & Singh, 1999). Figure 3 shows this overlay.

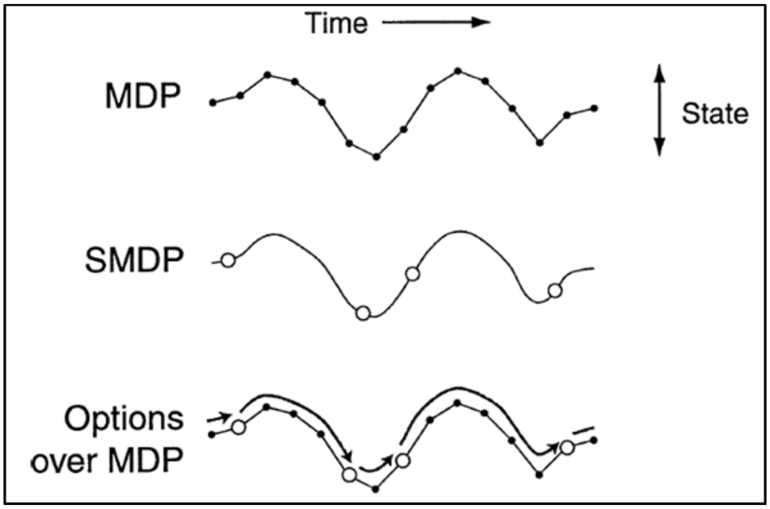

Figure 3 - Comparison among MDPs, SMDPs, and Options

2.2.5 Hierarchical Reinforcement Learning

The concept of plans and complex actions composed of simpler primitive actions is very powerful. As robust as RL is, it still has difficulty dealing with domains that have a high number of states. When the concept of composite actions is applied to reinforcement learning it opens the door to a methodology called hierarchical reinforcement learning (HRL). HRL gives regular RL the ability to scale up to large domains by allowing the domain's tasks to provide a context that limits the scope of its policies (Ghavamzadeh, Mahadevan, & Makar, 2006). This limitation occurs by breaking the overall task of the problem into subtasks, which are analogous to the plans described above. The subtasks become miniature SMDPs in themselves with their own local policies. A subtask is defined by the subset of states that are directly relevant to it. In this subset, there is the set of states from which the local policy can start, called the initiation set, and the set of states in which the local policy terminates, called the termination set. The subtask is also defined by the actions that can be chosen from states in this subdomain. These actions can either be primitive actions or other subtasks (Mehta, Tadepalli, & Fern, 2009) (Ghavamzadeh, Mahadevan, & Makar, 2006). Being able to call actions as either subtasks or primitive actions is what allows subtasks to be connected together in a hierarchical directed acyclic structure called a task graph. A task graph abstracts information not required for the completion of the present subtask by removing unnecessary states or state variables from consideration in the local policy. This is how HRL combats the problems caused by the curse of dimensionality.

Given a local policy for each subtask in the structure, a global hierarchical policy is formed for the entire problem. The execution of the global hierarchical policy is similar to a stack wherein each subtask is pushed onto the stack as it runs. If the current subtask invokes another subtask, then that child subtask in pushed onto the stack until it completes its run and then is popped from the stack after it terminates. If a parent subtask terminates for any reason, all child subtasks that are currently on the stack terminate and are popped as well (Ghavamzadeh, Mahadevan, & Makar, 2006). Each subtask is generally built around a sub-goal of the domain. Completing a subtask is thus like solving a component of the whole problem. Completing the root task is equivalent to solving the entire problem because in order to solve the root problem, you must first satisfy the sub-goals that it is made of (Mehta, Tadepalli, & Fern, 2009).

HRL easily expands from control of a single agent to control of multiple agents. Coordination can be effectively learned due to the hierarchical structure of the tasks. It allows agents to learn skills at the level of cooperative subtasks rather than only primitive actions. Cooperative subtasks can be used to dictate the jobs whose performance is improved when team work is involved. When they are placed high in the hierarchy of the task graph, they abstract the tasks that need coordination, making those tasks easier and faster to learn. This prevents the agents from getting muddied in low-level details found further down in the hierarchy that hinders progress. Levels of the task graph that have cooperative subtasks are called cooperation levels, and they define joint-action decision points. These joint spaces only include the local information of an agent and not the state information of other agents in order to limit the complexity of the decision process. The joint action space is defined by the union of the children of the cooperative subtask, be they subtasks themselves or primitive actions. Given just the actions of the other agent(s), an agent is capable of making competent decisions. Figure 4 shows the task graph of two robots that must cooperate to clean an office room. The root task of cleaning the room is a cooperative subtask because it requires cooperation between the robots to complete. A single robot must look at the child subtask (Collect Trash at T1 or T2) chosen by the other robot to decide the best course of action to take for itself. This level of the task graph becomes a joint-action space. One robot can only observe the actions taken by the other robot; it doesn't know its internal state (Ghavamzadeh, Mahadevan, & Makar, 2006).

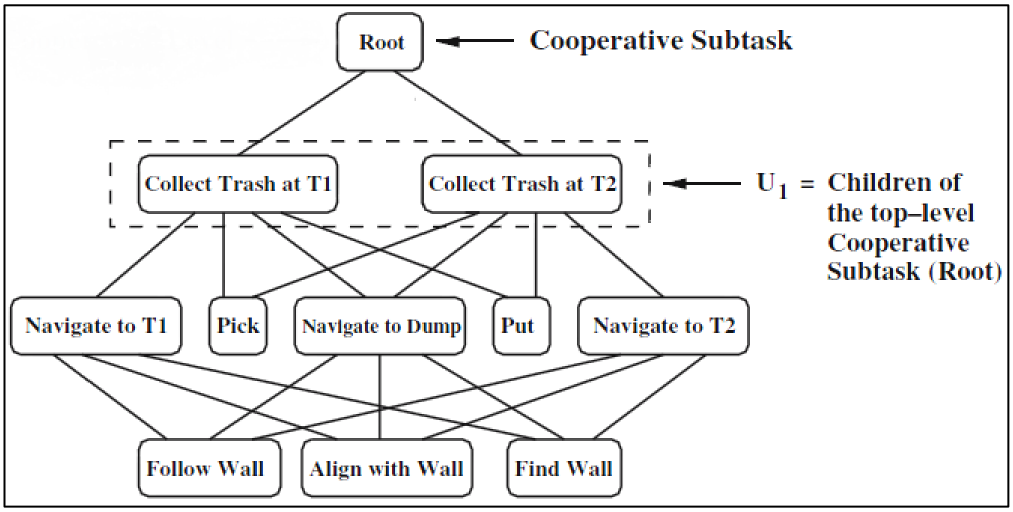

Figure 4 - Task graph of two robots that must cooperate to clean an office room

As stated before, the local policy of each subtask is modeled with a SMDP; cooperative subtasks are joint-action, so they are represented by multi-agent SMDPs. Agents attached to a cooperative subtask must choose extended actions that possibly depend on the actions of other agents. Because of the variable time scale of SMDPs, different extended actions will end at different times. The designer must determine when an agent can make its next decision in relation to when its chosen action completes and when the chosen joint action of the other agents in the cooperative subtask complete. There are three common strategies for deciding the next decision point τ-any, τ-all, and τ-continue. In τ-any, the next decision point is set as soon as any agent finishes its current action. The actions still being performed by other agents are stopped, and each agent selects a new joint action. τ-all waits for all agents to finish their jobs before the next decision point. Agents who finish before this time will stand idle until all actions are complete, and then all agents choose a new joint action. τ-continue lets agents make their next choice as soon as their current action finishes without interrupting other agents that haven't finished their task.

Once an agent completes an action, the next joint action is based on the current actions in the joint space that the other agents are still running. Both τ-any and τ-all are synchronous strategies because all agents must make a new decision together. This requires a decision making framework that can synchronize the agents at decision points. τ-continue is asynchronous because only a subset of agents choose their next action at each decision point (Ghavamzadeh, Mahadevan, & Makar, 2006). This thesis uses a form of the τ-all strategy. The lieutenants cooperate in completing their tasks, and before the captain can move to the next decision cycle and give the lieutenants another set of orders it must wait until they are both ready to receive new orders.

2.2.6 Belief, Desire, and Intention

Another methodology capable of high-level command and control is the use of Belief, Desire, and Intention (BDI) agents. Belief, desire, and intention are the three mental attitudes that a BDI agent has. Beliefs represent the information about the environment that an agent gathers, stores, and updates. They can be implemented as logical expressions (like a knowledge base), a set of variables, a standard database, or any other suitable data structure. There is a distinction between beliefs and knowledge. Knowledge is absolute whereas beliefs may be inaccurate because they only need to describe the probable state of the environment. Desires are the motivational part of an agent. They specify what the agent wants as well as their priorities or rewards and can be produced dynamically based on the current beliefs. Intentions are the deliberative portion of an agent. They incorporate part of the system's state to keep track of the present course of action that was selected to achieve the agent's goal(s) (Rao & Georgegg, 1995).

Just as in RL, BDI supports extended actions in the form of plans. Plans can have a hierarchical organization and are decomposable into subtasks and primitive actions (Rao, 1996). Also each plan centers on a goal and goals can be broken into sub-goals, each of which have an associated subtask or primitive action (van Oijen, van Doesburg, & Dignum, 2011). The system must be able to change plans on the fly, for example in the case that a particular action fails or an event in the environment occurs that forces a change in plans. A balance must be struck because excessive re-planning is too expensive but no re-planning is ineffective in a stochastic environment. The balance between responsiveness to the environment and determination to complete a goal lies in the notion of commitment. Commitment resides within the intentions of the system because an agent has no control over its beliefs or desires; however, an agent can manipulate its intentions. Each intention has two parts, a commitment condition that tells the state the agent wants to maintain, and a termination condition that tells when the agent should quit the current goal. Each intention boils down to a partially completed plan. In BDI methodology, plans are more than just a sequence of subtasks and primitive actions. They are defined by a body and a head. The body is made of the actions that must be performed for the plan to succeed. The head consists of the event that triggers the plan, and the precondition that specifies the situation that must be true in order for the plan to begin execution. This is analogous to the state an (S)MDP must be in before it can choose an action; the head gives context and dictates the options the system has (Rao & Georgegg, 1995).

Chapter 3 RELATED WORKS

3.1 Game Industry FPS AI

The following are examples of AI used currently by some of the popular commercial games in the industry. It illustrates how the use of techniques like FSMs is still prevalent.

3.1.1 Kill Zone

The Killzone franchise is a first person shooter game made by Guerilla Games for the Sony PlayStation. Its AI uses a technique called procedural combat strategies (PCT) which makes dynamic decisions for a single soldier, adjusting to the soldier’s combat situation at hand as well as its local terrain. The combat situation is the combination of the soldier’s threats and goals. After choosing a goal a soldier performs actions to achieve the goal until it is completed, it comes across a better goal, or the goal fails. For example a soldier chooses between attacking and positioning. This includes firing its weapon, selecting a point to get to, and navigating to that point. PCT uses an AI approach popular in games like chess called position evaluation functions (see Figure 5). Position evaluation functions are similar to fitness functions you find in evolutionary computation. They combine multiple properties of a particular goal into a single value with a weighted sum and use these values to compare the pros and cons of different goals. The properties reflect the competing importance of different aspects of the goal. Goals vary and are used in the game to define safe and dangerous spots during path-finding (see Figure 6), to decide the best position to throw a grenade, to classify enemy threats (see Figure 7), and even to prioritize other goals. The properties used as input to the position function are numerous and include things like proximity to desired location, cover from primary threat, the soldier’s line of fire, and preferred fighting range (Straatman, van der Sterren, & Beij, 2005).

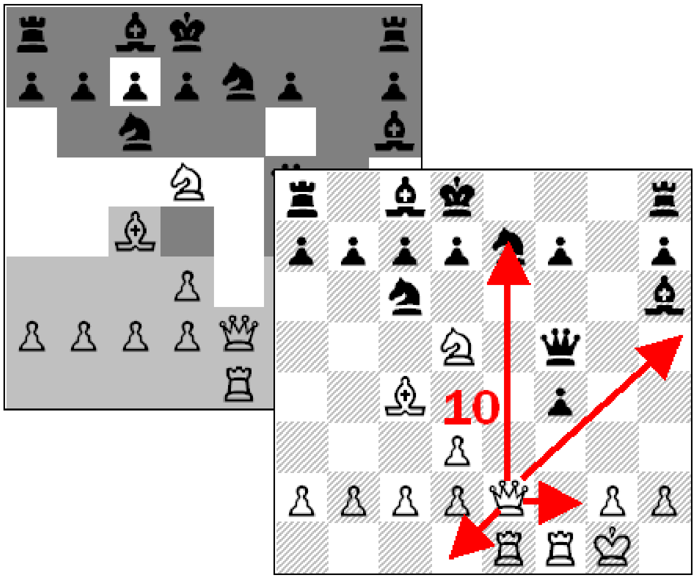

Figure 5 - Evaluation of different moves on chess board. Consider threats from other pieces.

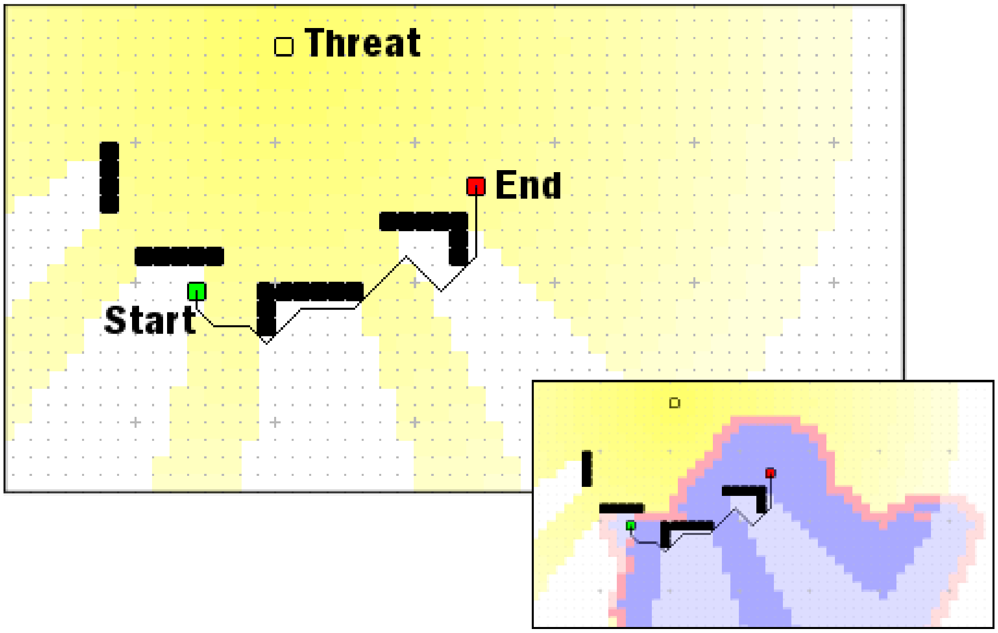

Figure 6 - Path finding evaluates positions' danger from threat, not just distance to goal.

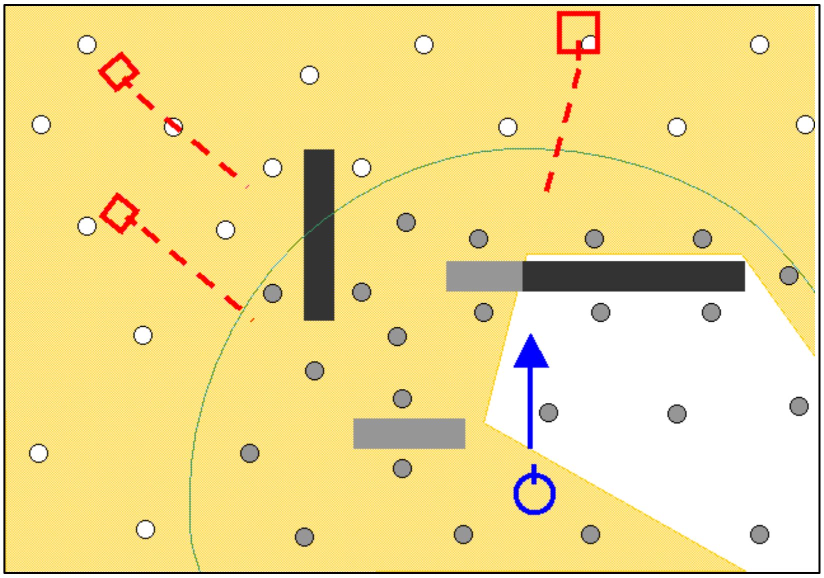

Figure 7 - Evaluation of different positions in game. Consider enemy lines of fire and mobility.

3.1.2 F.E.A.R.

F.E.A.R. is a first person shooter developed by Monolith Productions and employs conventional game AI techniques in an unorthodox way. It controls individual soldiers with a FSM and uses A* to plan a sequence of actions as well as for path finding. The FSM has three states called Goto, Animate, and UseSmartObject. Goto directs the soldier to a location, Animate performs an animation, and UseSmartObject performs a data driven version of Animate given context by an object (like turning a knob to open a door). Proper behavior modeling relies on the correct decision about when to switch between states and how to set the involved parameters for that transition. When the logic to change between the states is hardcoded, it becomes more complex and harder to manage as the FSM grows. Instead of implanting that logic directly into the FSM, F.E.A.R. lets a planning system maintain that code and dynamically chooses between states in real-time. It should be noted that the behaviors this technique models are not too advanced to be implemented with traditional methods. The point is that as the system grows, the interaction of the behaviors becomes increasingly sophisticated and unmanageable because the designer must explicitly create each detail. F.E.A.R. uses a real-time planning system in order to try to address this problem (Orkin, 2006).

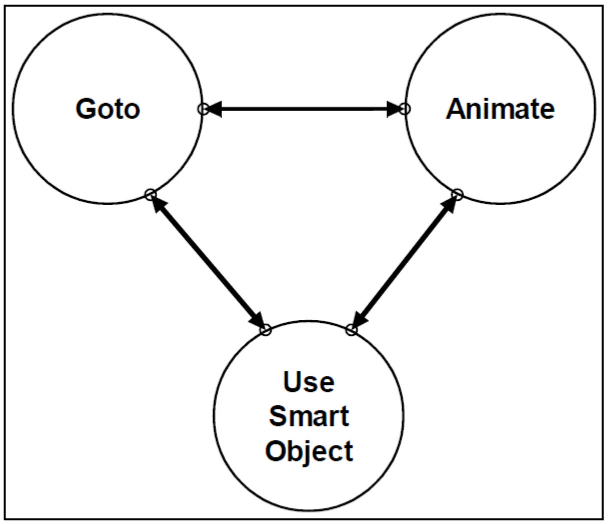

Figure 8 - Three state finite state machine used in F.E.A.R.

The FSM tells a soldier what it should do, but the planning system tells how to do it, providing goals to aid the soldier in figuring out the best course of action to achieve those goals. F.E.A.R.'s planning system resembles the STRIPS (Fikes & Nilsson, 1971) methodology in that there are preconditions that must be met before an action can be executed and there are post-conditions specifying how the environment should be changed after the action is completed. Soldiers are given goals to accomplish with different priorities. Soldiers accomplish goals in different ways depending on their action set. The action set determines what the soldiers can or can't do. The benefit of this approach is that goals are decoupled from the actions used to achieve them. If multiple soldiers are given the exact same goals but completely different action sets, they will display different behavior as they try to accomplish those identical goals. Adding additional behavior inside an FSM used to require that branches be added to it, but now F.E.A.R. only needs to add actions to the action set. Also because of the separation, new soldier types can be created by mixing goals and actions in different combinations. The modularity of goals and plans allows for the creation of atomic behavior that can be layered into more complex composite behaviors. The developer can put together goals and plans and let the planning system determine how they should interact, deciding how to transition from one behavior to the next (see Figure 9 & 10). Dependencies are resolved in real-time based on the current goal and the current action's preconditions and affects. The F.E.A.R. system allows for dynamic re-planning in case of the failure of the present plan. A soldier can try to achieve the same goal with alternative actions and behaviors when obstacles thwart previous attempts.

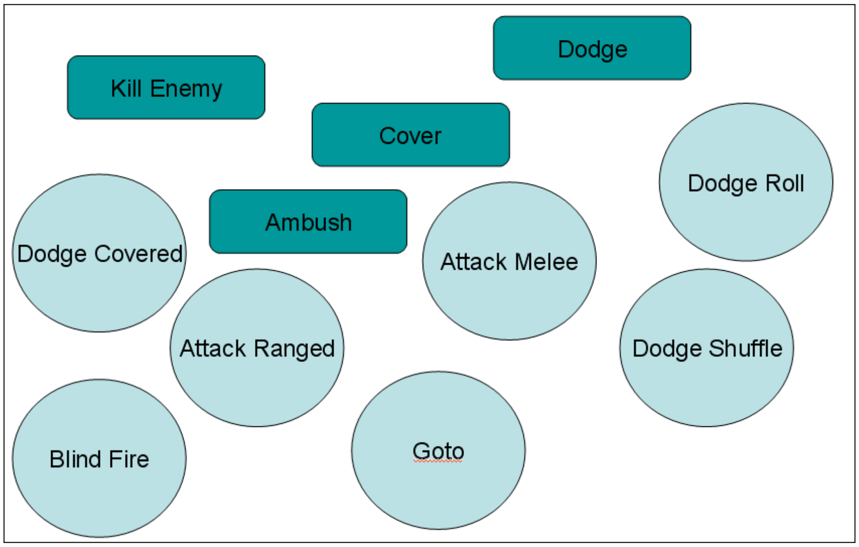

Figure 9 - Specify the goals (rectangles) and their actions (circles) independently of each other

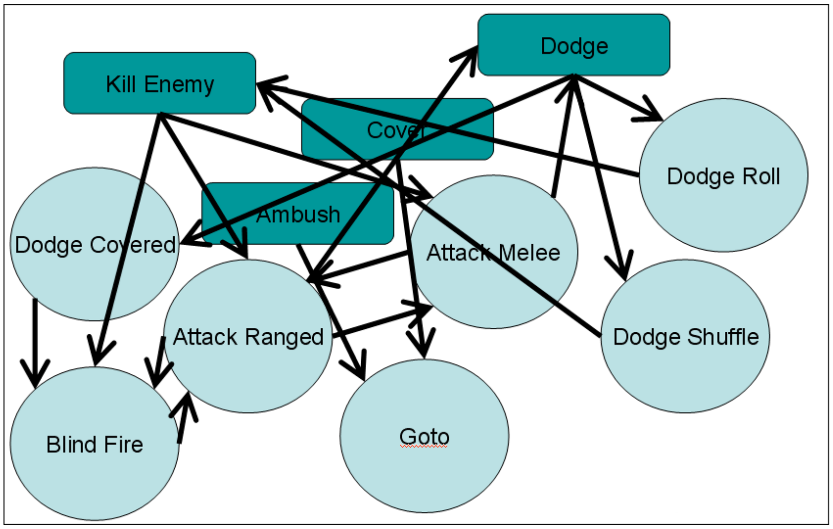

Figure 10 - Automating interaction between goals and actions.

F.E.A.R. planning is somewhat different from traditional STRIPS planning. F.E.A.R. assigns a cost to actions to influence the choice among multiple viable options. The inclusion of these costs is where A* comes into play. Although A* is typically used for navigation, in reality it is a general search algorithm designed to find the shortest path through a graph of nodes whose connecting edges have an associated cost. In the case of planning, the states are the nodes and the actions are the edges. A* thus finds the lowest cost sequence of actions that accomplishes the soldier's goal (Orkin, 2006).

3.2 Control of Individual Soldiers

There is research in the area of using RL to control individual soldiers. A large amount of soldier AI is controlled by FSMs. FSMs usually have static behavior and are difficult to maintain and update because of their hardcoded logic. Their strategies are fixed and do not improve as a result of gameplay (Patel, Carver, & Rahimi, 2011). Using Q-learning Patel et al develop a soldier meant to mimic the actions of an NPC in the video game Counter-Strike. They develop a soldier that learns to fight and compare its performance to soldiers hardcoded by humans. In Counter-Strike, planting a bomb at a specific point on the map is the main objective, not fighting the enemy. RL is used to reward completing this goal over killing enemy soldiers. They test the ability of a soldier to be pre-trained with an abstract model of the game environment containing a smaller amount of states than the actual environment (see Figure 11 & 12). They show that with this preliminary abstract knowledge, learning in the real environment can be accelerated and soldier performance is better in the initial stages of the simulation over soldiers whose learning starts from scratch. Their paper shows that RL can produce different behavior by changing the reward function and the utility given to different actions and that RL can evolve layers of behavior by adding more states to the learning process (Patel, Carver, & Rahimi, 2011).

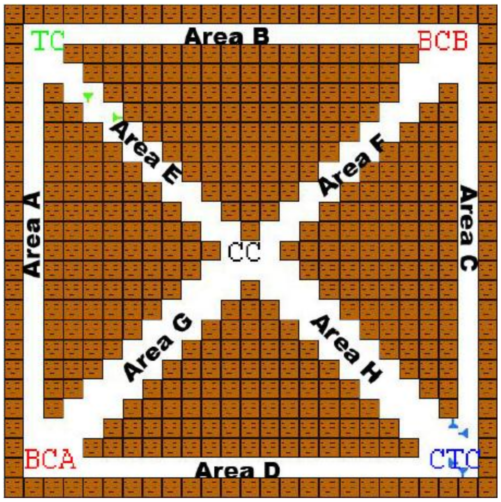

Figure 11- A simplified environment used to model Counter Strike

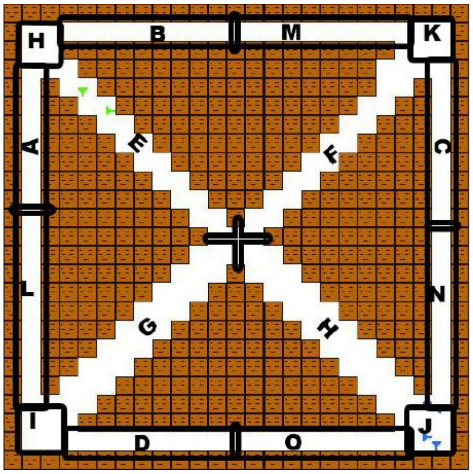

Figure 12 : A more complex environment used to model Counter Strike.

Instead of using Q-learning, McPartland and Gallagher use another RL approach called Sarsa(λ) to teach a soldier to navigate through a map, fight, and pick up inventory like ammo, three of the most common activities performed in FPS games. Sarsa(λ) is an algorithm similar to Q-learning in that it learns state-action pair values for the environment. An extra feature is that it uses eligibility traces which keep a history of the sequence of state-action pairs visited beyond just the most recent one. It gives a reward to each state-action value in the eligibility trace based on the current state in order to speed up the learning process. Changing the eligibility trace factor (λ) determines how far into the past a soldier can remember, and allows sequences of actions to be learned. McPartland and Gallagher again show that RL can be used to develop a controller for individual FPS soldiers, creating one controller for combat and another for navigation and inventory collection. They also show that RL is more efficient than rule-based systems because its underlying algorithm requires less code and needs less time for parameter tuning and updating (McPartland & Gallagher, 2008a).

In a later work, McPartland and Gallagher attempt to combine the two independent RL soldier controllers they developed before. They want to blend the smaller tasks already learned with RL into a larger more comprehensive behavior. The paper takes three different approaches to study the differences in behaviors that occur. The first combines the previously learned controllers through HRL. The second uses a rule-based reinforcement learner (RBRL) that decides when to switch between the previously learned controllers. The last combines the tasks of combat and navigation into a new bigger RL controller that has to learn the two tasks simultaneously. The HRL controller combines the states of the previously learned combat and navigation controllers. Its actions are subtasks made of the individual controllers themselves. The HRL controller has to learn when to switch between the smaller controllers based on the current state. The RBRL controller also uses the smaller controllers for its actions, and uses simple rules to switch between which one it executes. If an enemy is in site, the soldier uses the combat controller; otherwise it uses the navigation controller. The regular RL controller has the same state space as the HRL controller, but its structure is flat with no composite subtask for actions. Instead it has every possible primitive action from the smaller combat and navigation controllers meaning its state-action space is much larger than the other two combined controllers. The HRL and RBRL controllers outperform the flat combined RL controller. They act more like experienced players, but the RL controller plays like a novice sometimes shooting in panic in all directions. HRL seems the best strategy overall for its more diverse behavior during combat because its transitions between subtask controllers are not hardcoded by static rules like the RBRL controller (McPartland & Gallagher, 2008b).

3.3 Control of a Team of Soldiers

The approaches in the previous section control individual soldiers, but there are other techniques that attempt to control an entire team of soldiers simultaneously.

3.3.1 Hierarchical Task Network

The previous work shows that RL can be applied to the development of single agents for the creation of dynamic individual behavior. This thesis is more concerned with the coordination of a team of players, particularly in a game of domination. There has been work in recent years dealing specifically with this domain of multi-agent systems.

(Hoang, Lee-Urban, & Munoz-Avila, 2005) use a form of planning to direct soldier team behavior with a structure called a hierarchical task-network (HTN), which is very similar to the task graph used by HRL described earlier. HTNs break high-level tasks into a hierarchy of composite and low level actions. Like the task graph, an HTN has compound actions that are decomposed into primitive actions at the lowest level. The primary features of an HTN are its methods and operators. Methods specify the goal to reach, the preconditions to execute the plan, and the subtasks used to achieve it. Operators are primitive actions with post conditions that detail how the world should be changed. An HTN represents plans at the level of subtasks rather than at the level of primitive actions. They connect a team of soldiers controlled by an HTN to the video game Unreal Tournament (UT). The Simple Hierarchical Ordered Planner (SHOP) (Nau, Cao, Lotem, & Munoz-Avila, 1999) generates the plans that the HTN enacts. The problem with this approach is that the strategies are predefined and hardcoded. The first strategy attempts to control half plus one of the available domination points after which the soldiers patrol between the points and defend them. The second strategy tries to gain control of all the domination points (Hoang, Lee-Urban, & Munoz-Avila, 2005). These strategies are encoded to be tested for certain situations; no ML is employed in the process (Smith, Lee-Urban, & Munoz-Avila, 2007). Because of this lack of ML an HTN potentially requires its designer to be a subject matter expert on the domination gametype domain to know how to effectively combine actions hierarchically (Hogg, Lee-Urban, Munoz-Avila, Auslander, & Smith, 2011).

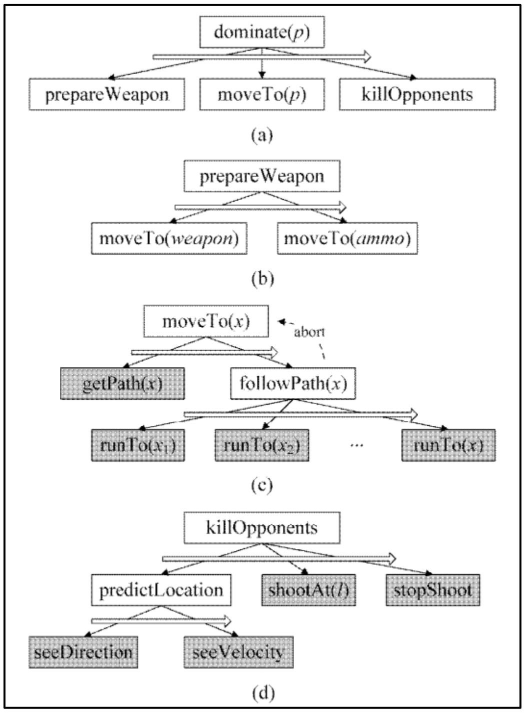

Figure 13 - Task decomposition of HTN

3.3.2 RETALIATE

(Smith, Lee-Urban, & Munoz-Avila, 2007) control a team of soldiers connected to UT like Hoang, but they do so with a RL system they call RETALIATE which stands for Reinforced Strategy Learning in Agent-Team Environments. Individual soldier behavior is static but interchangeable, meaning a developer can simply plug in an arbitrary soldier controller. For the experiment, soldiers are controlled by the same reactive state machine that comes standard with UT. This is done so that there is a fair comparison between team strategies and that one team does not have an advantage over another because their soldiers have superior individual behavior. This is also done so that the system can focus on team behavior without consideration for the complexities of individual soldier abilities. RETALIATE coordinates the team effort by pointing out important spots in the game environment, specifically the domination points. It exploits the fact that the game of domination inherently only rewards fighting around the domination points and not random areas across the map. Simply killing the enemy doesn't score a player points. However, if the player kills an enemy around a domination point, then that enemy is spawned somewhere else in the map, giving the player a chance to either capture the point or live on to continue defending it from further intrusion. RETALIATE learns a team policy using Q-learning, but removes the discount factor typically used; this causes the algorithm to focus on quick adaptation to an opponent's changing strategy rather than slow convergence to an optimal policy. RETALIATE's ideology is that optimality is not necessary, only fast exploration towards a winning policy (Smith, Lee-Urban, & Munoz-Avila, 2007).

RETALIATE's state space is defined by the different domination points in the map and which team, if any, owns them. If there are three domination points and two teams, then each point can be owned by either team. There is also the special case at the start of a match where none of the points are owned by either team. The state can thus be represented by a tuple (P1,P2,P3) where each P stands for a domination point and can take on the value C (captured), L (lost), N (neutral). For example the state (C,N,L) means the first domination point has been captured, the third has been lost to the opposing team, and the second one has not been taken by either team. This means RETALIATE has a problem space of size $(T+1)^D$ where T (equals 2) is the number of teams and D is the number of domination points on the map. It can execute team actions which are actually joint actions of individual soldier actions. An individual soldier action consists of going to a specific location in the map. So each action performed by the algorithm tells the entire team of soldiers to go to different locations based on the current state. This can pose a problem because the number of team actions grows exponentially with the number of players on the team. Each soldier can potentially be sent to each domination point. This means for a team of size M, there are M^D different joint actions, where again D is the number of domination points. Another potential problem is that the locations of domination points are not known by the RETALIATE soldiers beforehand. Exploration of the map and discovery of the points must happen before learning can proceed, which slows the learning process. The utility of an action is determined directly by the state of the world afterward. It is calculated by subtracting the number of domination points lost from the number of domination points captured. So the state (C,C,L) has a value of 1 because there are two captured points and one lost point. It would be of higher utility than the state (N,L,N) which has a value of -1 because there are no captured points and one lost point (Smith, Lee-Urban, & Munoz-Avila, 2007).

3.3.3 CBRetaliate

The work done in RETALIATE is extended by Auslander et al by combining RL with case-based reasoning. The original RETALIATE takes a little too long to adapt against an opponent that changes strategy. They create CBRetaliate to try to remedy this issue. It uses features of the game to calculate similarity metrics to compare the current situation with past ones. These features include the number of team members, each team's score, how far team members are from domination points, and which teams own what point. Cases are stored in a knowledge base, and each case has a set of these similarity features with unique values as well as their own Q-table. When it is time for comparison, a case will be retrieved if the current game features sufficiently match the features of that stored case. CBRetaliate runs the original RETALIATE underneath everything, but when there is a successful match it switches to the previously learned Q-table associated with that past case. If there are no matches found during comparison against the stored cases, a new case is created and stored. A case can only be created if the team is beating its opponent and is scoring points at a faster rate than the enemy team. Meeting this condition implies that the Q-table being used is employing a winning strategy and should be saved for later use. The problem with CBRetaliate is that when its opponent constantly changes strategy, CBRetaliate becomes no more effective than the original RETALIATE algorithm. Both algorithms are designed to deal with an opponent that changes its strategy then uses that same strategy for a period of time before changing again. They simply can't adapt fast enough to deal with an enemy that constantly changes strategy with rapid succession (Auslander, Lee-Urban, Hogg, & Munoz-Avila, 2008)

3.3.4 RL-DOT

RL-DOT (Wang, Gao, & Chen, 2010) uses an RL approach similar to that of RETALIATE to direct of team of agents in a match of domination in the video game UT. RL-DOT uses a boss NPC to directly control multiple soldier NPCs. The boss NPC tells them which domination points to either attack or defend, similar to RETALIATE; however, the actions of RL-DOT are not as explicit as actions in RETALIATE. The actions of RL-DOT are just abstract intentions of how the system should distribute its soldiers across the domination points. This means it comes up with a number of soldiers that should be assigned to each point, but it does not actually determine which soldiers go where. Because each soldier can be sent to each domination point, there are a variety of ways to explicitly execute the boss NPC's abstract intentions, some better than others. The interpretation of these abstract commands is handled in a lower level of the hierarchy. Like RETALIATE, all individual RL-DOT soldier controllers are identical. This simplifies the RL algorithm because it doesn't have to account for differences in individual soldier behavior. There is no consideration for how differences in offensive and defensive behavior can completely change the way a team plays the game. The downside of this approach is that it may simplify the model too much. Unlike RETALIATE, RL-DOT soldiers are aware of the location of the domination points beforehand which cuts down on slow exploration. It is also common in video games for human players to have some a priori knowledge of the map such as the location of domination points in the case of the domination gametype (Wang, Gao, & Chen, 2010).

The key difference between the two is that RETALIATE does not account for the effect of enemy activity on the game while RL-DOT does. A conventional MDP works best in an environment where only it can affect the state transition, meaning only its actions cause the environment to move from one state to another. However UT, as all other FPS games, is a game where at least two opposing teams impact the same environment, meaning the actions of the enemy cause the environment to change state as well. This alters the way the MDP learns and RL-DOT makes up for this deficiency by modeling enemy activity in its algorithm. The UT video game gives its soldiers the ability to report what enemy it sees to the system and where it saw that enemy. The algorithm uses this information to keep track of each enemy team member and their relative location. By tracking enemy movement, RL-DOT maintains an approximation of the opponent's plans by assuming that a soldier is attacking the domination point that it's closest to. These observations of enemy position are incorporated into the decision making process and used to update the Q-learning table appropriately. The states of the learning process are determined by the number of domination points, and which team owns each of them, just like for RETALIATE, but the utility of a state is determined only by the number of domination points RL-DOT owns (Wang, Gao, & Chen, 2010).

RL-DOT faces the possibility of poor performance with a large number of team members just like RETALIATE. It mitigates this risk by having its actions be represented by soldier distributions across the map. It just gives the number of soldiers that should be sent to each domination point and leaves it to another algorithm to figure out the details of the deployment as opposed to giving explicit instructions about where each individual soldier should go. Despite this, both approaches have one soldier in charge of both offensive and defensive deployment of soldier agents, and all those agents exhibit the exact same behavior.

This thesis differentiates itself from previous work by splitting up the responsibility among a single captain and two lieutenant agents, one in charge of offense, and the other defense. This divides the task of learning where to send soldiers as well as how many to send. It will make the task of soldier deployment cooperative, allowing each lieutenant to be charged with taking care of a smaller task, but at the same time working together to accomplish the overall goal. Soldiers assigned under the lieutenants will also display different behavior depending on whether they are playing offense or defense. As a result, the team will have the ability to showcase a wider variety of behaviors. This approach will scale better as the number of soldiers grows because it reduces the state-action space and abstracts certain data in the decision process. Both RL-DOT and RETALIATE tested five or fewer soldiers on one team. This thesis will test ten soldiers per team, which is common for the number of members on a team in modern games. It doubles the size of the team controlled in these other approaches but without exponentially increasing the size of the state-action space, which RL-DOT and RETALIATE would have to do.

Chapter 4 HQ-DoG METHODOLOGY AND IMPLEMENTATION

4.1 Simulation Environment

The simulation environment is built with a 3D game engine called Unity and is based off of the domination gametype found in the Call of Duty franchise. Figure 14 shows the simulation environment with obstacles and the domination points. HQ-DoG starts all of its games near point C and the enemy it competes against starts games near point A. A navigation mesh is placed on top of the map so that soldiers can move from one position to another using A*. Teams poll the environment every two seconds to determine targets which they do and don’t own, and every five seconds they get one point for each target they possess. There is a five second cycle that respawns all dead soldiers. There is also a ten second cycle that removes all dead bodies from the map so that they do not pile up and get in the way of living soldiers as they move around.

Domination is a gametype that rewards teamwork. The more soldiers that attempt to attack a domination point together, the less amount of time it will take to capture that point. To begin the capture process, a soldier only needs to stand inside the capture radius of its target. It takes a single soldier ten seconds to capture a point by himself, so there is a time bonus for each soldier that helps him capture that point, but only up to five soldiers. This means that if five or more soldiers attack a point together, it will only take one second to capture. The time bonus is limited to five soldiers so that the capture process does not happen too quickly, otherwise it would only take one-tenth of a second to capture a point if the entire team were to take advantage of the time bonus.

Each domination point has five respawn points associated with it that are placed within its vicinity. If a soldier is killed during the game it is respawned randomly at one of the respawn points connected to a domination point its team owns. If one of the teams does not own any domination points, because either there are neutral positions that it has not captured or the other team has captured all the domination points, then a soldier will be respawned randomly from any of the respawn points on the map.

This setup rewards a team for maintaining control of a position because if a soldier is killed in the process of defending a domination point it is more likely to respawn near that point. If the soldier is respawned near the point, it has a better chance of making it back to continue to defend that point as opposed to being respawned somewhere far away across the map.

If a team is able to keep consistent control of two of the domination points, say A and B, then its soldiers will continue to respawn near those targets when they die and the enemy soldiers will only be able to respawn at point C. This is a phenomenon known as Spawn Trapping because unless the team that only owns one domination point is able to capture another point, its soldiers are effectively trapped continually respawning at point C. If the team that owns points A and B continues to hold just those two points and doesn't try to capture all the points, the spawn-trapped team will be at a constant disadvantage if they are unable to escape from the trap because they will only score points at half the rate of the other team until the end of the game. This is an effective way to play the game, but it is a strategy that requires teamwork. In a way this mechanism also punishes greedy teams because if a team captures all three points, the enemy soldiers will respawn randomly all across the map and not near a specific area, making their movement harder to anticipate and counteract.

It is important to note that this simulation is not a complete recreation of the domination gametype found in Call of Duty, just a simplified model. It does not include features like the ability to through grenades and smoke screens, call in special vehicles to attack the other team, have enhanced physical abilities, or obtain special attachments for weapons. All soldiers have the same physical abilities and use the exact same weapons. It is not the purpose of this thesis to build a fully functioning game, only to show HQ-DoG's ability to learn a competitive policy against various opponents.

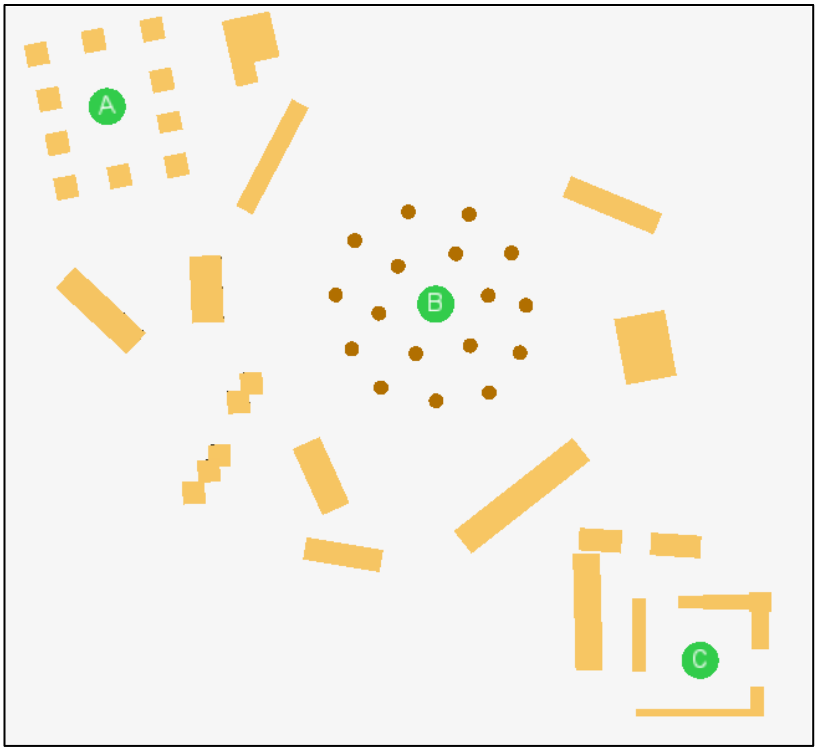

Figure 14 - The domination map used for HQ-DoG

Figure 15 shows a diagram of the flow of the HQ-DoG algorithm. There are three major components which include:

- Captain's Process – Captain senses the game environment, updates Q-table based on utility, chooses action, and then gives orders to lieutenants.
- Lieutenants' Process – The lieutenants each interpret their separate tasks from the captain's orders, update their observation models and Q-tables, and choose action for soldier deployment.
- Deployment of Soldiers – The lieutenants each send their actions to an algorithm that calculates the most efficient way to execute soldier deployment across the map based on distances of the soldiers from different domination points.

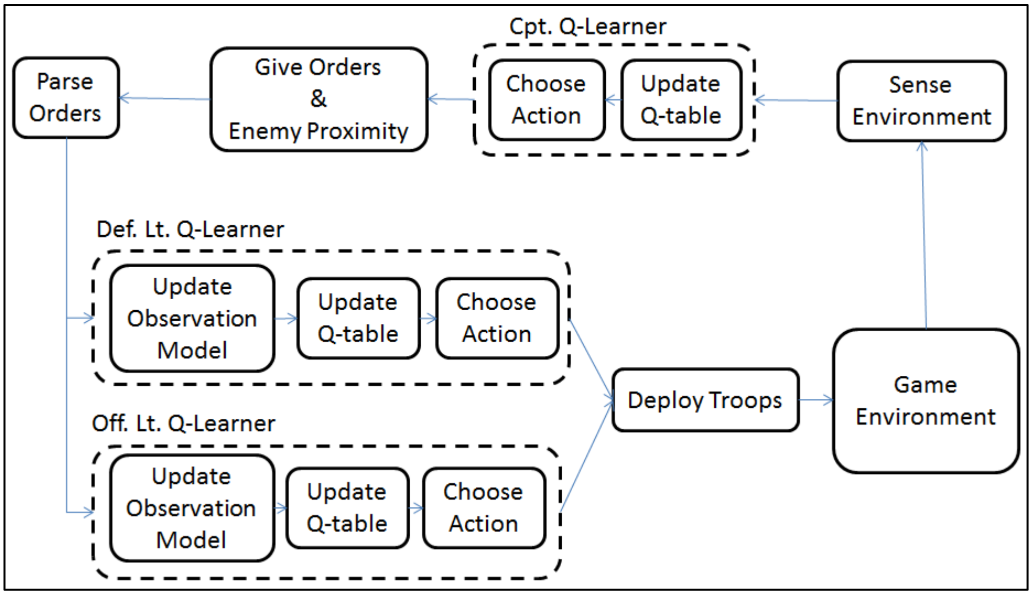

Figure 15 – Steps of the HQ-DoG algorithm starting from sensing the game environment

4.2 Captain's Process

The captain begins the decision making process. It is his job the decide the overall strategy of the team for the current decision cycle.

4.2.1 Sense Environment

The captain waits for notification from both of its lieutenants that they are ready to receive their next order then it senses the current state. The state of the captain is directly influenced by the game environment. HQ-DoG determines state the same way as RETALIATE and RL-DOT, by the status of ownership for each domination point. There are three domination points each of which can be owned by one of the two opposing teams or neither of them, thus ownership status can take three values. This means that there are a total of twenty-seven states that the captain can sense from the environment. A single state is described by a tuple of three values from the set (Lost,Neutral,Captured). The captain's state-space remains the same size regardless of the number of soldiers on its team. Appendix A shows the captain's entire state space.

4.2.2 Choose an Action

After sensing the environment the captain must choose its next action based on the current state. Each decision cycle it chooses an action that captures, defends, or ignores each of the domination points, so a single action consists of a tuple of three values from the set (Capture, Defend, Ignore). The captain has to balance exploration of new parts of the state-action space with exploitation of the Q-values it has already learned from its experience and interaction with the environment. This means deciding whether to choose the best learned action for the current state as of yet or choosing an action other than the best to see if it yields a better utility. Q-learning settles this by choosing the best action each decision cycle with probability ε and choosing a random action with probability $(1 - \varepsilon)$. There are twenty-seven states and twenty-six possible actions for the captain which means it would have a state-action space of 702 if it could execute each possible action from every state.[1] However, HQ-DoG reduces this state-action space by restricting what qualifies as a valid action for each state. The set of valid actions become a function of the state.

HQ-DoG soldiers behave differently based on whether they are attacking or defending a domination point. If the soldier is an attacker, it goes to a point and remains there until the point is captured. If the soldier is a defender, it patrols around the perimeter of the target looking for enemies trying to capture the point it’s defending. Each decision cycle the captain chooses an action that attacks, defends, or ignores each domination point in some combination. Because of how soldiers carry out orders based on their type, it is ineffective for the captain to choose an action that defends a point it doesn’t own or attack a point it has already captured. Therefore, given the current state, any action that attacks a Captured position or defends a Neutral or Lost position is invalid. This approach reduces the number of valid actions for each state from twenty-six to seven and decreases the captain’s state-action space to 189. Appendix A shows the captain’s state-action space with the IDs of each state’s valid actions.

[1] There are twenty-six actions and not twenty-seven because the action [Ingore,Ignore,Ignore] ignores all domination points and does nothing to protect or increase the team’s score.

4.2.3 Calculate Utility and Update Q-table

Once the captain has performed its chosen action, it is able to determine which domination points it owns and which it doesn't and it uses this to calculate the utility of the current state. As in RETALIATE utility is calculated as the difference between the number of points owned by the teams. Specifically the captain subtracts the number of points owned by the opposing team from the number of points it owns. Calculating utility like this gives a range of values from +3 (when the captain owns all three domination points) to -3 (when the opposing team owns all points). After the captain has the utility for the current state it uses that utility to update the Q-value of the previous state-action pair with the standard equation for Q-learning (see Algorithm 1in Section 2.2.3).

4.2.4 Give Orders and Proximity

The captain passes the same global action to both of its lieutenants that dictates which targets to attack, defend, or ignore. Along with the global action, the captain gives the lieutenants the observed proximity of enemy forces. HQ-DoG attempts to model the actions of the enemy soldiers because they affect the state as well. HQ-DoG's soldiers tell the captain the location of enemy soldiers when they encounter them. However unlike RL-DOT, HQ-DoG does not try to account for the proximity of each and every soldier on the enemy team. As the number of soldiers per team increases, the number of possible observations for enemy activity as defined in RL-DOT increases exponentially. Instead HQ-DoG informs the lieutenants of whether or not there are enemies near any of the domination points with a true or false value. Along with decreasing the observation model, this approach is also closer to the way information is passed when actually playing the domination gametype. When a player communicates enemy activity to his teammates, he is more likely to tell what point they're closest to or whether or not there are any enemy soldiers at a specific point. He won't give details as to the exact number of enemy soldiers. The captain is able to give information about enemy activity because when its soldiers encounter the enemy, they pass the enemy soldier's position up the chain of command. The captain maintains a record of each enemy soldier position by tracking which domination point the soldier is closest to. When the next decision cycle comes the captain abstracts this information out to whether or not the enemy is near each point, meaning there are eight different enemy approximations. This abstraction allows the information used by the lieutenants to be simplified and reduces their observation models and Q-tables. It also scales well to domains with larger team sizes because the eight possible approximations will not change with team; they will only change with a different number of domination points.

4.3 Lieutenant's Process

It is the job of the lieutenants to take their orders from the captain and work together to collectively accomplish their tasks. Offense and defense are decoupled which makes it possible to learn the tasks separately.

4.3.1 Interpret Orders

Because each lieutenant is given the same global action from the captain, they must both interpret the global action to derive their own individual orders. A lieutenant's state space is determined by the number of targets it's been assigned and the percentage of the team's soldiers it has control of to accomplish its task. The problem is to find the right balance between the two as there is a limited supply of soldiers and it would be of little use for the captain to decide to attack all three domination points and then give the offensive lieutenants no soldiers to carry out its orders. HQ-DoG borrows the concept of supply and demand from economics to find this balance. Figure 16 shows a picture of the general idea. As the number of domination points designated for capture increases the demand for offense rises as well, and the demand for defense falls; the opposite of this is true as well. As more points need to be defended, the system responds by supplying the defensive lieutenant with more soldiers. The double-sided arrow in the diagram depicts this trade off. Moving right along the arrow signifies a more defensive strategy, and moving left represents a more aggressive strategy. The diagram is not completely accurate though, as it does not account for when one or more points are ignored by the captain and are neither attacked nor defended. Appendix B gives a full chart of how the lieutenants derive the percentage of forces they received based on the captain's orders and how that translates into hard numbers given that there are ten soldiers on the team.

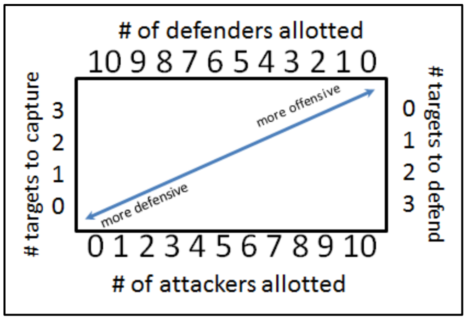

Figure 16 - Lieutenant Soldier Allotment vs. Captain Target Decision

4.3.2 Choose an Action

When a lieutenant chooses an action, it is confined within the orders it is given by the captain and dependent on the number of targets is has been assigned. Its actions pertain to a distribution of soldiers across the targets it must defend or attack. Appendix B lists all the possible distributions a lieutenant can choose based on its orders and what that means in terms of concrete numbers. These distributions are predetermined by the designer but are based on reasonable assumptions about how to deploy forces effectively. If the captain chooses to attack all three domination points (with a team size of 10 soldiers), it would not make much sense to send eight soldiers to the first target and only one soldier to each of the remaining two targets. The captain has the option to only attack one point and ignore the others if it chooses. If the captain decides to attack all three targets it's more sensible to devise distributions for the lieutenant that commit forces to the targets in a manner that gives the plan of action a reasonable chance of success. This is the philosophy that guided the creation of these distributions.

Like in the captain's Q-learner, the lieutenants choose the best learned action with probability $(\mathbf{1}-\boldsymbol{\varepsilon})$ and choose a random action with probability $\boldsymbol{\varepsilon}$. This approach solves the problem of trying to learn the utility of each possible permutation of deployment for soldiers across three targets which grows exponentially with team size. The use of a set of percentages can be applied to multiple team sizes and does not grow exponentially as the number of soldiers grows. There are eight different distributions a lieutenant can choose from, but only a subset of these can be chosen from a given state. The eligible subset is a function of the current state. Specifically it depends on the number of targets it's been assigned; the lieutenant can't send its soldiers to only two positions when it's ordered to attack or defend three. Figure 17 illustrates how a different set of actions A_i can be chosen based on the current state. Appendix B shows the subset of actions a lieutenant can choose from given the current state. Given how a lieutenant's states, observations, and actions are defined, it has a state-action space of 122. Algorithm 2 on page 52 shows how action selection works.

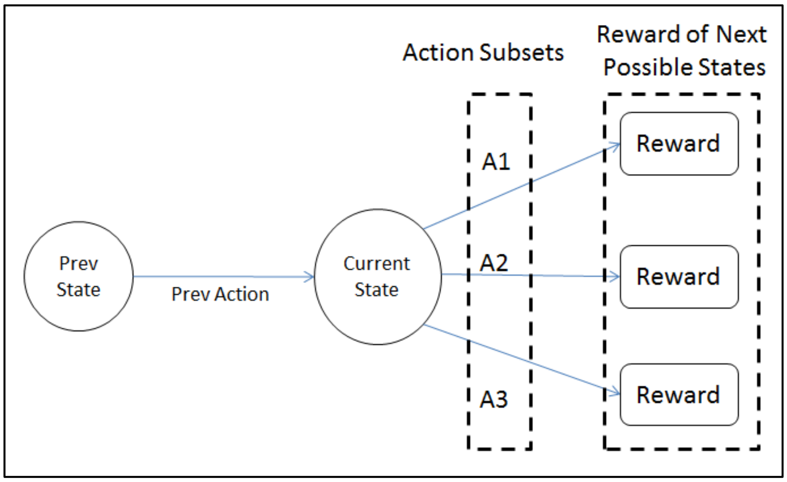

Figure 17 – Different subset of actions A_i can be chosen based on current state

4.3.3 Update Observation Model

The captain passes its observation of the enemy location in its global action order, but it does not use the observation itself in its decision making. The approximation is meant for the lieutenants to use when updating their observation model, which is done in a manner similar to what happens in RL-DOT. Appendix B shows the different observations that can be made when a lieutenant is assigned one, two, or three targets. The observation model consists of state-observation pairs and connects a probability $O(s, o)$ with each pair. It tracks how often different observations have occurred in a certain state and adjusts their associated probability accordingly. Each time an observation is made, the probability for that observation is increased and the probabilities for all other observations in the state are decreased by an equal amount so that the total sum of the probabilities still equals one. The more often an observation is made, the more likely it is that that observation is a true representation of the enemy's current activity. Algorithm 2 below details how the observation model is updated for each lieutenant.

4.3.4 Update Q-Table

A lieutenant's Q-table, unlike a captain's, incorporates observations into its Q-factors where each factor consists of a state, observation, action triplet $Q(\langle s, o \rangle, a)$. Because of this, the equation for updating the Q-table is modified. HQ-DoG uses an equation defined by RL-DOT for calculating the maximum future reward for a Q-table that incorporates observations. The main difference is that determining long-term reward for the modified equation requires finding the Q-factor whose action maximizes value over all observations for every possible executable action. The state and observation components of a lieutenant's Q-factor separately form a state-observation pair that has a probability associated with it in the lieutenant's observation model. The value of each Q-factor that an action is a part of is multiplied by the probability of the state-observation pair that is also part of the Q-factor. All these products are then added together, and the action with the highest sum is returned as the maximum future value for the current state. Example 1 shows how to calculate maximum future reward for the Q-factor where lieutenant has been assigned all three targets, is allotted the entire team to carry out its task, and the lieutenant chooses action 0. The lieutenants use the same reward function as the captain when updating their tables. Algorithm 2 gives the equation for how this process works.

Example 1 – Calculating Expected Maximum Future Reward for Lieutenant

- For each state s, action a and observation o
- State: Assigned all three targets ([ABC]) and allotted entire team (100%)
- Action ID: 0 (33%, 33%, 33%)
- Max future reward = $max_a = \sum_o [O(s,o)Q(\langle s, o \rangle, a)]$ =
  ([ABC] 100%, [TTT])([ABC] 100%, 0, [TTT])
  + ([ABC] 100%, [TTF])([ABC] 100%, 0, [TTF])
  + ([ABC] 100%, [TFT])([ABC] 100%, 0, [TFT])
  + ([ABC] 100%, [TFF])([ABC] 100%, 0, [TFF])
  + ([ABC] 100%, [FTT])([ABC] 100%, 0, [FTT])
  + ([ABC] 100%, [FTF])([ABC] 100%, 0, [FTF])
  + ([ABC] 100%, [FFT])([ABC] 100%, 0, [TFT])
  + ([ABC] 100%, [FFF])([ABC] 100%, 0, [FFF])

Algorithm 2 Q-Learning Algorithm for Lieutenants

1. For each state count the number of different possible enemy observations that can be made $|\boldsymbol{E}|$. Also initialize a counter k for the number of times this state visited to zero.
2. Initialize the probability of each observation to $\frac{\boldsymbol{1}}{|\boldsymbol{E}|}$ for the state it is a part of.

```
3. Initialize each state, observation, action triplet to zero
4. Choose action $a$ after receiving orders from captain
        • $a \leftarrow \arg max_a\, Q(\langle s,o\rangle,a)\ with\ probability\ (1-\varepsilon)$
        • $a \leftarrow random\ action\ with\ probability\ \varepsilon$
5. Perform action $a$
6. Observe current state s.
7. After observation $o'$ made in state $s'$, increment k for observation $o$ in state $s'$ and update model.
   For each observation o in s:
        • $O(s',o) = \frac{k\,O(s\prime,o)\,+\,1}{k+1}\ if\ o = o'$
        • $O(s',o) = \left(\frac{k}{k+1}\right) O(s',o)\ if\ o \neq o'$
7. Update Q-table for previous state
        • $Q(\langle s,o\rangle,a)\ = (1-\ \alpha)Q(\langle s,o\rangle,a) + \alpha[r +\ \gamma\ max_{a\prime} \sum_o O(s',o)Q(\langle s',o\rangle,a')]$

8. Transition state to $s'$.
```

4.4 Soldier Deployment

After both lieutenants have decided how many soldiers they want to go to each target HG-DoG must determine the most efficient way to incorporate their actions together. No soldier is permanently assigned to a lieutenant for the entire duration of a match. The structural hierarchy has to change and adapt according to the captain's orders for different types of behavior. This means that the algorithm must be able to assign any soldier to either of the lieutenants during each decision cycle. This assignment is based on which domination points have been selected as targets, how many soldiers need to go to each target, and how close a soldier is to each of the targets. HG-DoG uses what RL-DOT defines as an action interpretation to find this most effective deployment. An action interpretation takes the combined actions of the lieutenants and sends soldiers to different targets based on their minimal distance from each point. Once a certain target has reached the quota of soldiers it needs to fulfill its part of the combined action, it rejects all other soldiers even if they are closest to it. Soldiers are then assigned to their next closest target until that target reaches its quota. In doing this HG-DoG ensures that soldiers are not traveling longer distances across the map than they need to. This minimizes the time it take HG-DoG to execute a single action, which is very important because domination is a time sensitive game. The algorithm will continually lose if it's unable to react to changes in the environment in a timely manner. Algorithm 3 describes soldier deployment.

Algorithm 3 Soldier Deployment Across Map

1. Initialize a distance matrix whose elements are the distance of each soldier from each domination point. Each row i represents a soldier, each column j represents a target.
2. The quota for each domination point in the combined distribution is quota$_n$ where $n \in (1,2,3)$ stands for the number of soldiers needed for the 1st through 3rd targets.
3. Sort the elements of the array from smallest to greatest distances.
4. Select the smallest element from the matrix
    - If the element's row has already been assigned a target or the element's column has already reached its quota, move on to the next smallest element.
    - Otherwise assign the soldier to this target, mark the soldier as assigned, and increment the number of soldiers currently committed to this target. Also assign the soldier to the proper lieutenant so it knows what its behavior for this cycle will be.
5. Repeat step 4 until the end of the sorted list is reached.

4.5 Different Soldier Behavior

At any given time a soldier can be either an attacker or defender dependent on whether it is assigned to the offensive or defensive lieutenant. Attackers and defenders have different behavior to accomplish different tasks. Attackers are meant to capture points and defenders are meant to protect them.

4.5.1 Attacker Behavior

If a soldier is assigned to the Offensive Lieutenant and given a point to capture, it becomes an attacker and will take the shortest path to its target. If it comes across any enemies along the way it will engage and shoot at them until they are dead or out of range before continuing on to its target. If it arrives at its target, an attacker will remain there until it has successfully captured the point at which time it will report to its assigned lieutenant that it has completed its mission. After an attacker captures the target it was assigned, it will then proceed to attack the closest neutral or lost point on the map until it has been issued new orders or killed. If an attacker is killed en-route to or while it is in the process of capturing a target, it will report to its lieutenant that it has been killed in action and respawn at another randomly selected location after a certain amount of time. After it has respawned, an attacker will continue with its original mission, if its target still has not been captured, until it's given new orders.

4.5.2 Defender Behavior

A soldier becomes a defender after being assigned to the Defensive Lieutenant. Each domination point has a set of waypoints that form a path around its perimeter. When a defender is given a domination point to defend, it finds the closest of these waypoints and takes the shortest path to it. Like the attacker, it engages any enemies it confronts along the way until they are dead or out of view. It reports it has been killed in action if it dies before reaching its target. Once at the closest waypoint, a defender follows the path the waypoints create, patrolling around its target, warding against potential attackers from the opposing team. After it starts patrolling, a defender reports to its lieutenant it has begun doing so. If while patrolling, enemy forces are able to get past the defense and being capturing the domination point, a defender goes to the point and tries to stop them. It attacks the intruders until they are all dead or it is killed. If the intruders were able to capture the point a defender is tending, it goes to the point and attempts to recapture it; after recapturing it goes back to patrolling around the perimeter. A defender repeats these actions until it is killed or given new orders. If it is killed, it will respawn somewhere else and continue on with its mission.

Chapter 5 RESULTS AND DISCUSSION

5.1 Opponent Strategies

To evaluate its efficacy, HQ-DoG is tested against seven different opponent strategies broken into three categories to determine if the algorithm can learn a competitive policy. The first category has one strategy that uses a team whose soldiers all attack the closest un-captured (neutral or lost) domination point. The second category has five strategies each of which is static, meaning the strategy directs each soldier to go to a specific domination point. The soldier will repeatedly go to that same point throughout each match. The final category uses one strategy that chooses randomly between three different static strategies each game match. These strategies themselves are non-learning; they do however represent the range of sophistication and skill with which the domination gametype can be played.

HQ-DoG is not tested against RETALIATE nor RL-DOT because their state-action spaces would be too large if they were expanded to accommodate the control of a team with ten soldiers. The captain of HQ-DoG has a state-action space of 189 regardless of the size of the team. For a team of ten soldiers a single lieutenant has a state-action-observation space of 122. Combining the Q-tables of the captain and both lieutenants, HQ-DoG has a state-action-observation space of only 433 for a team of ten soldiers. For a team of ten RL-Dot has 27 states, 66 possible actions, and 66 possible observations giving it a state-action-observation space of over 117,000. RETALIATE would have a prohibitively large state-action space as well. They both would both grow exponentially with larger team sizes.

5.1.1 Greedy Strategy

The Greedy Strategy is the strategy used in the first category and is one of the simplest of the seven strategies. Each soldier in this strategy continually runs towards the closest un-captured domination point and remains there until the point has been captured or it is killed. There is no coordination whatsoever among the soldiers; they all act completely independently of each other. This strategy is used for testing because it is actually what one is most likely to encounter in the real world. In online multiplayer shooter games that have the domination gametype each player on the team is free to act and make decisions for himself. He can attack or defend whichever target he decides is most advantageous regardless of what his teammates are doing. There is no central force commanding the team which is compounded by the fact that most people playing online are absolute strangers randomly put together on a team. Although there is the ability for players to talk to each other if they have a headset, this feature is very rarely used for effective coordination. If anything communication might be used to notify other teammates of the whereabouts of the enemy. For the purposes of this thesis, such autonomy implies very dynamic and unpredictable team behavior because everyone on the team is acting on their own accord. The extreme lack of coordination, however, can be problematic as the behavior is purely reactionary. It does not anticipate enemy actions at all.

5.1.2 Static Strategies Category

A Static Strategy utilizes the exact same strategy each game match. In many ways it is the complete opposite approach to the Greedy Strategy. It may seem counter-intuitive to do the exact same thing throughout the match, but in domination it can lead to an advantage (in the right circumstances) because of its consistency and coordination. Consistency and coordination alone are not enough though. They must be used together with a smart deployment of soldiers. Not every static strategy intelligently directs it soldiers, and the second category exhibits five different static strategies that range from effective to ineffective in their deployment. If a static strategy for domination is defined by a triplet that denotes the number of soldiers sent to domination points A, B, and C in that order, then the five Static Strategies in the second category are (5,5,0), (4,4,2), (3,4,3), (6,2,2), and (8,1,1).

5.1.2.1 Static Strategy (5,5,0)

The Static Strategy (5,5,0) devotes half its forces to constantly attack domination point A and the other half to attack domination point B. This is arguably one of the best strategies for playing domination because it focuses on capturing just two of the three points on the map and then maintaining control for the duration of the match. It essentially tries to spawn trap the enemy (see Section 4.1). The team does not have to maintain possession of all three points on the map to achieve an advantage because doing so is likely to spread its forces thin. The team only needs to consistently keep control of two of the points in order for its score to grow faster than the opponent's. Given the choice to control the same two points the entire match or controlling different pairs of points, it is better to maintain control of the same two points. Otherwise the team has to capture a point previously owned by the opponent, which typically takes more effort and time than capturing a neutral point. In most domination maps, points A and C are positioned near the edges close to where the two teams start off, and point B is positioned near the middle. As a result it is usually better to try to control points A and B together or points C and B together because points A and C are far apart. The Static Strategy (5,5,0) attempts to unswervingly control points A and B and take advantage of their proximity. This is the type of behavior one might expect from a group of people consciously playing together as a team. Each player is given a role to play and they all perform their jobs in concert, determined to collectively stick to a single course of action. It is a more complex strategy than running to the nearest un-captured point like the Greedy Strategy. The downside of this strategy is that it completely ignores one of the domination points, and if the team is unable to maintain control of either of the two points it has committed its forces to, it will lose.

5.1.2.2 Static Strategy (4,4,2)

The Static Strategy (4,4,2) is close to the strategy (5,5,0) and attempts to capitalize on its benefits by constantly focusing on just two of the domination points. However, (4,4,2) tries to mitigate the pitfalls of (5,5,0) by sending two soldiers to the third target so that the opponent team cannot keep control of it uncontested. The two soldiers are not really meant to capture point C; this size of force is too small to have a chance of contending against any serious defense. The two soldiers are only meant to pester the enemy team and cause them to devote at least some minimal amount of force to defending C, which detracts from attacking either points A or B. The downside of (4,4,2) is that it must spread its forces across the map to do so.

5.1.2.3 Static Strategy (3,4,3)

The Static Strategy (3,4,3) spreads its soldiers as evenly as possible across all three domination points, but because there are ten soldiers, one of the targets will get more soldiers than the other two. This strategy tries to maintain control of the middle of the map by devoting more forces to point B. This is a good idea because it is usually harder to maintain control of point B than any other point given its position in the center where the most foot traffic will be. Sending an equal amount of force to the remaining two positions attempts to maintain control of the target closest to the team's starting point while also applying pressure on the opponent to defend the target closest to its starting point and not give it a free pass. This strategy is very aggressive because it wants to capture everything, but there is the chance that it can be too aggressive. By spreading its forces into groups to attack all the points simultaneously, it is also reducing the number of soldiers that can be in each group; these forces might get spread too thin. Also the strategy is obviously not trying to take advantage of the Spawn Trapping phenomenon because if it is successful in capturing all points, the enemy will spawn randomly at all respawn points across the map.

5.1.2.4 Static Strategy (6,2,2)

The Static Strategy (6,2,2) is a defensive approach to playing the domination gametype. It focuses on maintaining a defensive posture around point A and making sure it never loses it by devoting more than half the team to protecting it. Of course by doing this, the strategy surrenders the majority of the map to the enemy. It commits an even amount of force to the other points, but this only results in two soldiers left for each of the remaining groups. Because of its high degree of defensiveness it is not expected that this will be a competitive policy. In domination it is not enough to hold just a single point. If a team wants to be defensive, it is usually only wise to be defensive after at least two points are held. Strategy (6,2,2) doesn't retreat completely into a shell around point A, but the forces it sends to points B and C cannot hope to compete against the enemy. At best the soldiers sent to points B and C might be able to capture their respective targets if there is no enemy force present for a long enough period of time. They might gain a few points by doing so, but the they have no chance of defending either point B or C and will be overwhelmed the moment HG-DoG commits forces to attack those points.

5.1.2.4 Static Strategy (8,1,1)

The Static Strategy (8,1,1) takes all key points of Strategy (6,2,2) to the next level. It is protective of domination point A, but to a degree that it is too defensive. The enemy would have to essentially commit the entire team to the offense of point A if it wanted to capture it, but that would be unnecessary as it could just attack and hold points B and C. The strategy only sends one soldier to the point B and C each, which is a very ineffective use of its forces. Of all the Static Strategies in the second category, it is predicted this one will perform the worst because it does not efficiently deploy its soldiers across the map to find the right balance of offense and defense.

5.1.3 Random Strategy

The Random Strategy is similar to the Static Strategy in that it chooses one strategy and continues to employ that strategy for the duration of the game match. The difference from the Static Strategy is that the Random Strategy chooses randomly among competitive static strategies each game match. The chosen static strategies are determined by which of them win the most games against HQ-DoG in their separate experiments. The Random Strategy is somewhat of a compromise between the Greedy and Static strategies. On one hand, it tries to capitalize on the consistency of the Static Strategy, but on the other it changes the strategy every game match so that it is more difficult for HQ-DoG to learn an effective policy to counter it.

5.2. Experimental Results

Each experiment consists of five hundred separate game matches. The first team to reach a score of 100 wins the match, but in the rare case that both teams reach 100 at the same time, the match ends in a draw. The captain and lieutenant Q-tables are initialized at the beginning of the experiment. The Q-tables update as games progress through matches and they pass on what they have learned to the next game until the end of the experiment. Experiments are independent of one another, so all Q-tables are reinitialized at the start of a new experiment.

The number of game matches per experiment was determined by looking at how experiments in similar work were conducted. In other papers researching the domination gametype experiments had up to

200 game matches for testing a team of five soldiers. Because this thesis doubles the number of soldiers on a team from five to ten, it increases the number of game matches in order to give the algorithm more time to learn. This is also the reason why the score required to win a game match is increased to 100 compared to only 50 in other papers. A higher score limit causes games to run longer, giving the algorithm more time to learn within a single match. More time is needed for HQ-DoG compared to other approaches because even though HQ-DoG uses a hierarchy and data abstraction to reduce the domain's state-action space, it is still directing twice the number of soldiers controlled by other algorithms. The larger team size naturally increases the size of the state-action space which makes the learning process for HQ-DoG more difficult.

5.2.1 HQ-DoG vs. Greedy Strategy.

Figure 18 shows the win/loss chart for HQ-DoG against the Greedy Strategy. A green bar (top half) signifies when HQ-DoG won the match and a red bar (bottom half) means the Greedy Strategy won. Contiguous bars of the same color mean that consecutive games were won. A space with neither a green nor red bar means the matched ended in a draw. As can be seen in the chart, HQ-DoG dominated the Greedy Strategy for the majority of the trial. Of the five hundred games, the Greedy Strategy only won twenty-nine matches. As was stated before, the Greedy Strategy is a very simple approach to playing domination. It requires no communication or any coordination among the soldiers; as such it is a strategy that only reacts to changes in the environment. HQ-DoG is able to both react and plan in the environment by learning when it's better to be offensive versus defensive. Positioning defenders around a target plans against potential future attackers from the opposing team, which is something HQ-DoG learns to do.

The environment is stochastic, and executing even the most competent plan of action at any given time is not guaranteed to succeed because in the end it comes down to whether a team's soldiers can survive long enough to perform their tasks. This is why the Greedy Strategy is able to win even in late stages of the experiment. Despite this, it appears that executing a plan with coordination and communication like HQ-DoG is far superior to a strategy without it such as the Greedy Strategy. In this experiment HQ-DoG wins 94.2% of the matches. Figure 19 shows

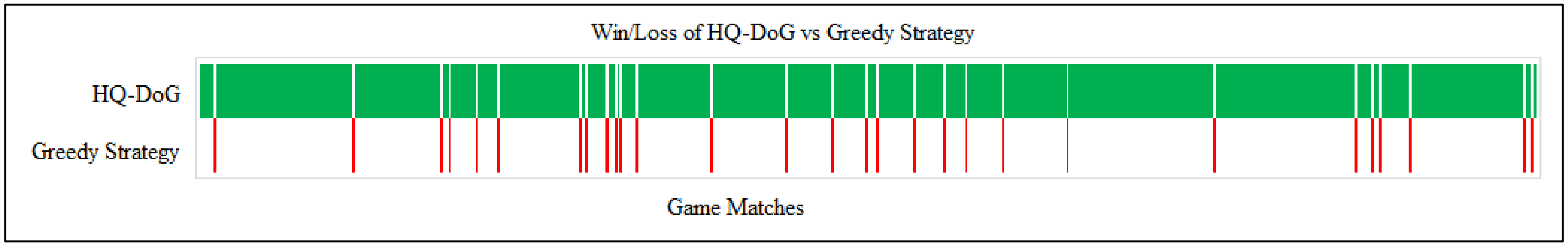

Figure 18 - Win/Loss chart of HQ-DoG vs. Greedy Strategy

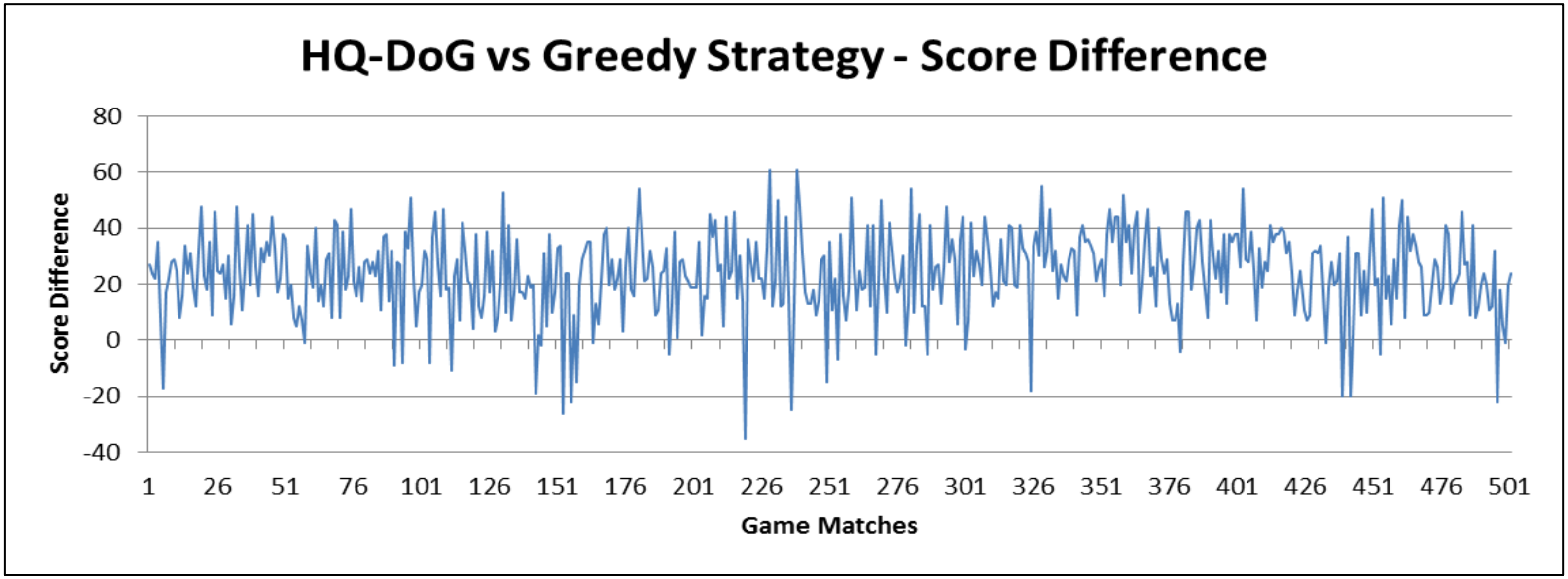

Figure 19 - Score Difference of HQ-DoG vs. Greedy Strategy

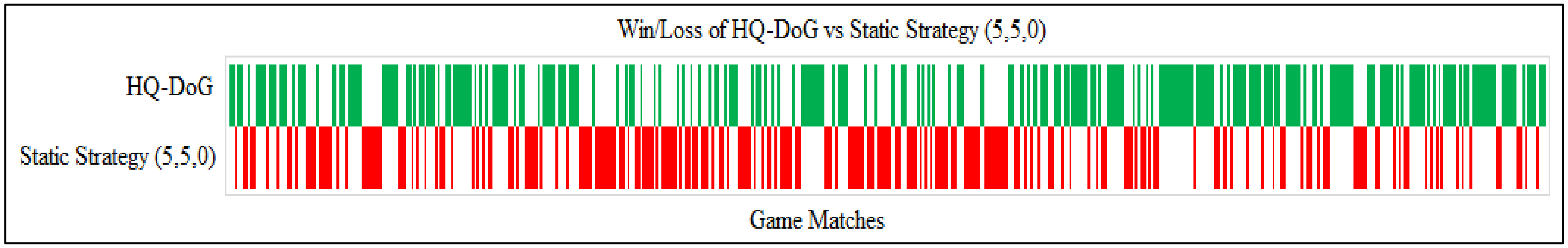

Figure 20 - Win/Loss chart of HQ-DoG vs. Static Strategy (5,5,0)

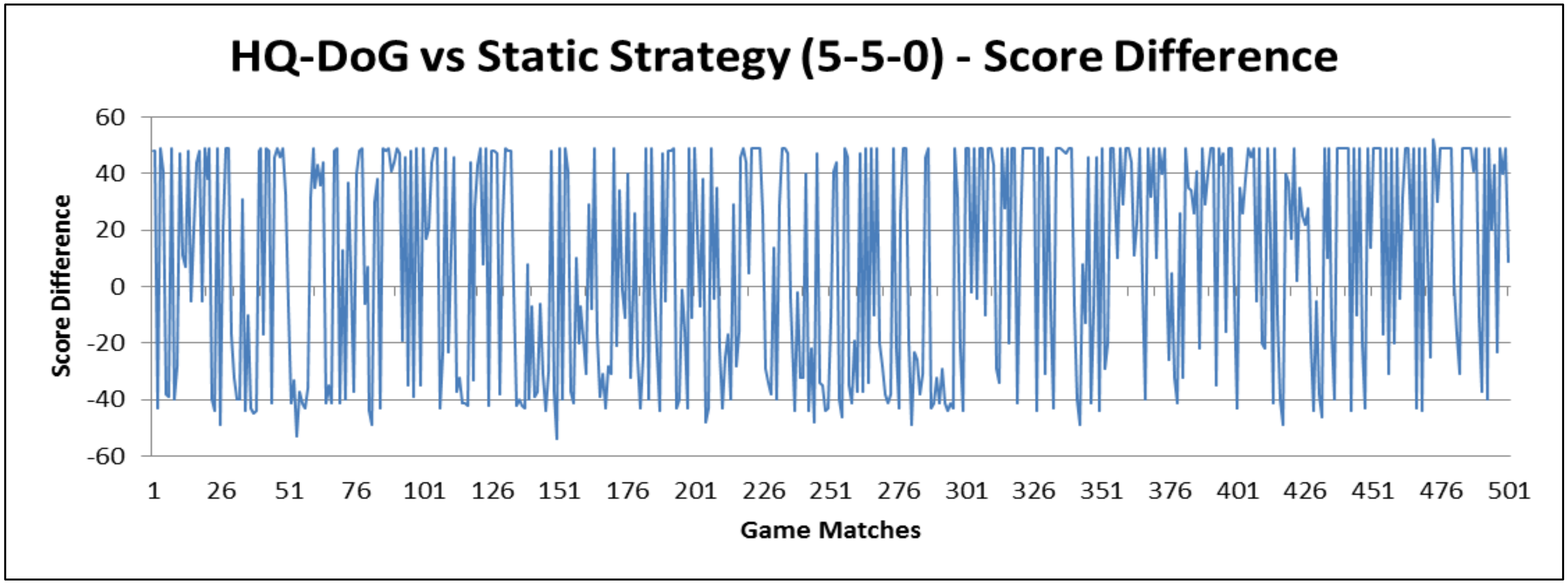

Figure 21 - Score Difference of HQ-DoG vs. Static Strategy (5,5,0)

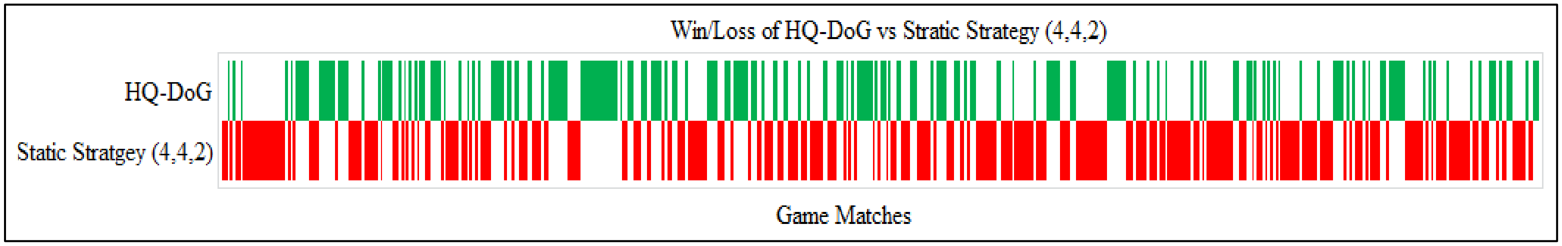

Figure 22 - Win/Loss chart of HQ-DoG vs. Static Strategy (4,4,2)

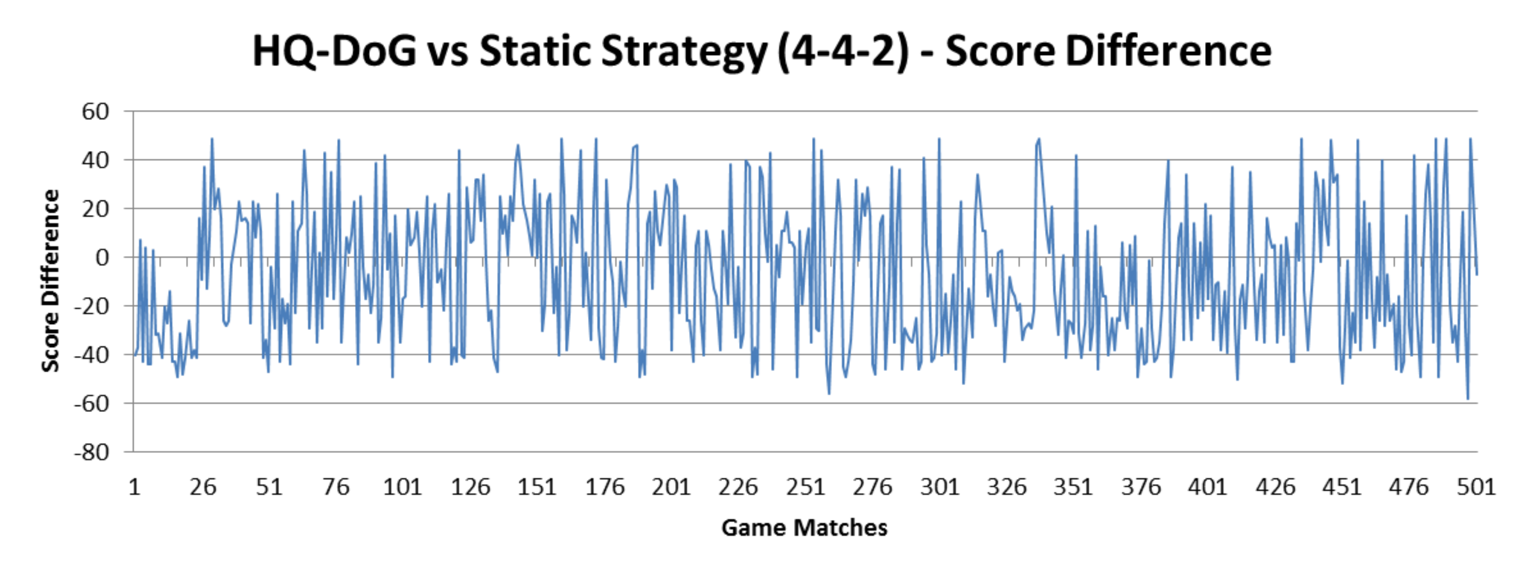

Figure 23 - Score Difference of HQ-DoG vs. Static Strategy (4,4,2)

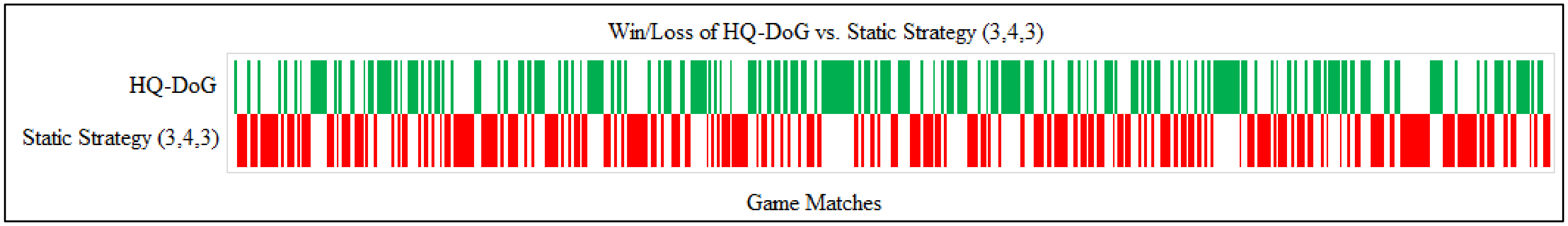

Figure 24 -Win/Loss chart of HQ-DoG vs. Static Strategy (3,4,3)

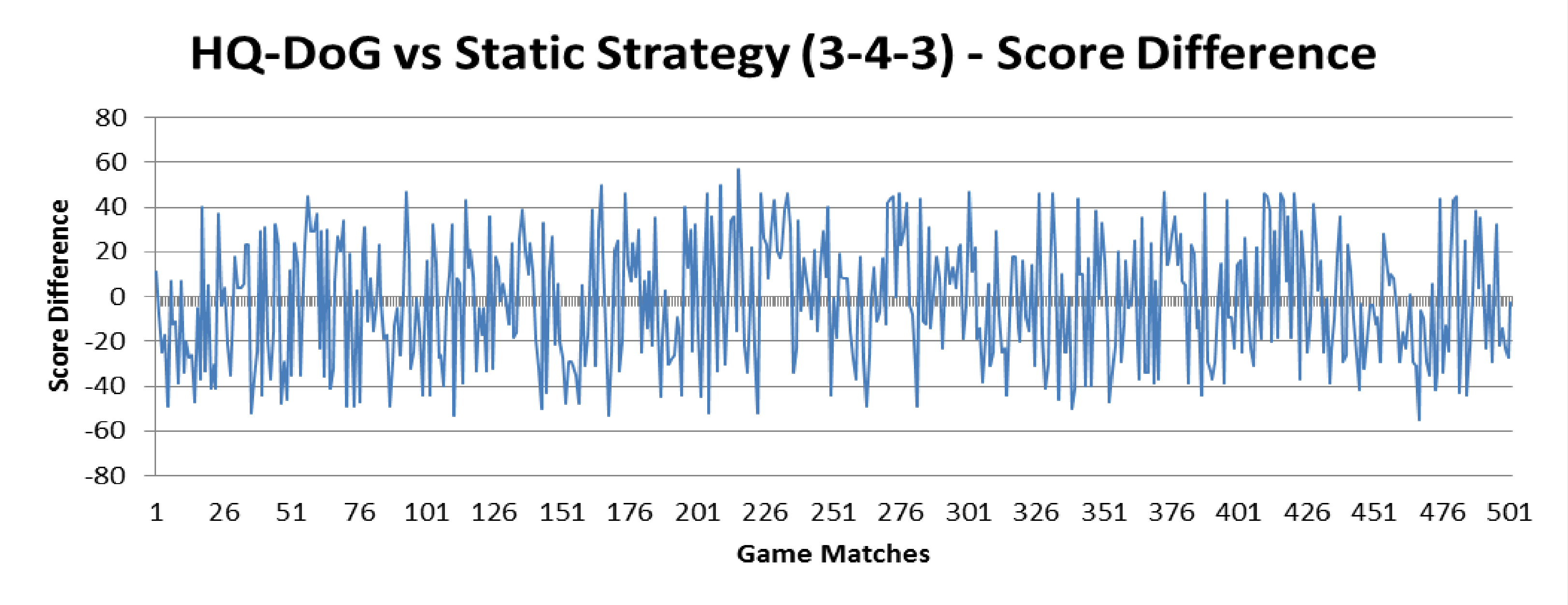

Figure 25 - Score Difference of HQ-DoG vs. Static Strategy (3,4,3)

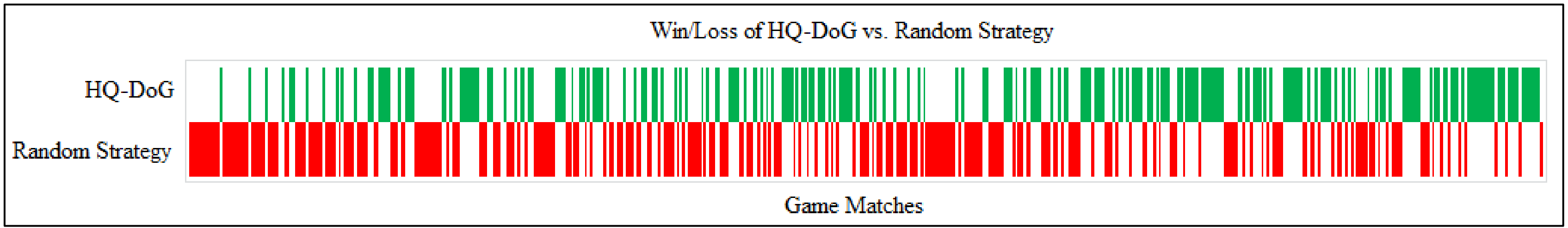

Figure 26 - Win/Loss chart of HQ-DoG vs. Random Strategy

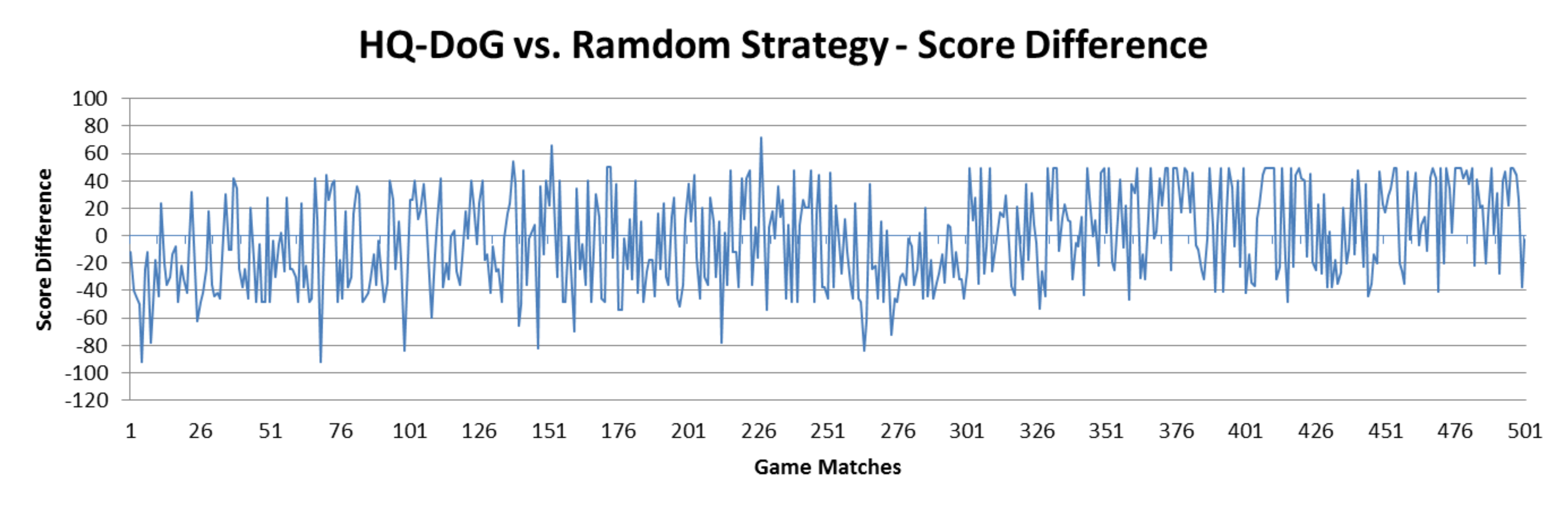

Figure 27 - Score Difference of HQ-DoG vs. Random Strategy

a line graph of the difference in scores for each game of the experiment. A positive difference indicates HQ-DoG had a higher score, negative means the Greedy Strategy had a higher score, and a difference of zero indicates a tie. It shows that for most of the matches, HQ-DoG wins by an ample lead. The average lead over the Greedy Strategy is 23.794.

5.2.2 HQ-DoG vs. Static Strategies

One of the benefits of testing against Static Strategies is that it is known in advanced exactly what the strategy is trying to accomplish every moment of any given match. This fact makes it easier to analyze the HQ-DoG Q-tables when trying to determine if they successfully learned the action to execute to counteract the static strategy.

5.2.2.1 HQ-DoG vs. Static Strategy (5,5,0)

Figure 20 shows the win/loss chart of HQ-DoG against the Static Strategy (5,5,0). It is obvious that HQ-DoG has a much more difficult time competing against the Static Strategy (5,5,0) compared to the Greedy Strategy as indicated by the fact that it wins far fewer games. The Static Strategy (5,5,0) focuses on attacking only domination points A and B, and HQ-DoG must learn that it doesn’t have to devote forces to the defense of point C. Table 1 shows that when HQ-DoG only owns point C and has lost the other two points, it learns it’s best to attack point B with the full force of the team. This overwhelms the force of the enemy at point B, which is likely to be five soldiers at the most. The action associated with this, [Ignore,Attack,Ignore], has the highest Q-value of -2.78004. Table 1 also shows that the algorithm learns the worst thing it can do in this situation is only defend point C. This action, [Ignore,Ignore,Defend] has the lowest associated value of -5.76506. Similarly Table 2 displays the portion of the captain’s Q-table for the state when all domination points are neutral (the very beginning of the game). By the end of the experiment the captain learns that the best action is to attack points B and C as its first move. The action that accomplishes this, [Ignore,Attack,Attack], earns a Q-value of 8.119135 which is much higher than the values of the other actions that can be executed from this state.

The captain learns to attack and defend the appropriate positions when necessary, but this alone is not enough. The lieutenants must learn to send the appropriate amount of force to their targets. The Static

Strategy (5,5,0) commits five soldiers to both points A and B, and this is a very formidable force to deal with. It only takes three or four shots to kill a soldier with full health, so it is nearly impossible for a soldier to survive against an enemy that significantly out mans it. If Static Strategy (5,5,0) has a hold on either of A or B, HQ-DoG must devote more than half the team to attacking the point if it hopes to have any chance of capturing it.

The algorithm learns how much force to use upon observation. Table 3 shows the offensive lieutenant learns to send half the soldiers to point C, the point closest to its starting position, at the beginning of a match and send the rest of the team to point B. Sending five soldiers to C captures the point faster than sending three soldiers would. If it sent seven to C that would only leave three soldiers to capture point B, which would most likely fail against the enemy's five soldiers. After capturing C these soldiers join the rest of the team trying to capture point B, providing reinforcements, and this is usually successful. The max Q-value of 1.519102 associated with that action reflects that the lieutenant learns.

By learning these maneuvers HQ-DoG is able to become competitive by the end of the experiment. Overall, it wins 54.8% of the matches. Figure 21 shows the difference in scores during the course of the experiment, the average lead of which is 8.656 for HQ-DoG. It is clear the graph straddles zero more than the score difference graph for the Greedy Strategy meaning HQ-DoG lost more games against Static Strategy (5,5,0). However it also means HQ-DoG won many games, over half, and that it was not overpowered. This shows that even against one of the best overall strategies, HQ-DoG is able to learn a competitive policy from scratch based solely on its experiences.

Table 1 - Captain Q-table vs. (5,5,0). Lost point A & B. Captured point C

State	Action	Q-Value	Visits
Lost,Lost,Captured	Attack,Attack,Defend	-3.64959	212
Lost,Lost,Captured	Attack,Attack,Ignore	-3.39036	454
Lost,Lost,Captured	Attack,Ignore,Defend	-4.03783	280
Lost,Lost,Captured	Attack,Ignore,Ignore	-4.18537	465
Lost,Lost,Captured	Ignore,Attack,Defend	-3.71928	285
Lost,Lost,Captured	Ignore,Attack,Ignore	-2.78004	742
Lost,Lost,Captured	Ignore,Ignore,Defend	-5.76506	170

Table 2 - Part of Captain's Q-table vs. Static Strategy (5,5,0)
State: All Points Neutral (beginning of game).

State	Action	Q-Value	Visits
Neutral,Neutral,Neutral	Attack,Attack,Attack	5.696432	150
Neutral,Neutral,Neutral	Attack,Attack,Ignore	1.690666	4
Neutral,Neutral,Neutral	Attack,Ignore,Attack	2.898006	4
Neutral,Neutral,Neutral	Attack,Ignore,Ignore	-0.014946	6
Neutral,Neutral,Neutral	Ignore,Attack,Attack	8.119135	322
Neutral,Neutral,Neutral	Ignore,Attack,Ignore	2.726068	5
Neutral,Neutral,Neutral	Ignore,Ignore,Attack	4.590344	9

Table 3 - Lieutenant's Q-table vs. Static Strategy (5,5,0).
State: Assigned to attack Points B and C. Given 100% of team. No enemy seen at either Point.

Assignment	Observation	Force	Distribution	Q-Value	Visits
B C	False,False	100%	[7,3]	-0.165	8
B C	False,False	100%	[5,5]	1.519102	172
B C	False,False	100%	[3,7]	-0.04839	62

5.2.2.2 HQ-DoG vs. Static Strategy (4,4,2)

Static Strategy (4,4,2) focuses on capturing domination points A and B and pestering its enemy at point C, causing the enemy to devote forces to its defense. Figure 22 displays the win/loss chart of HQ-DoG against the Static Strategy (4,4,2) and shows that HQ-DoG had more difficulty against it than against Static Strategy (5,5,0) as it only wins 41.4% of the matches. Figure 23 displays the score difference between the two and shows that the difference is negative more than it is positive. The average difference is -7.138. Static Strategy (4,4,2) outperforms Static Strategy (5,5,0) because it does not ignore domination point C; two soldiers can capture a point given a long enough absence of the enemy. Even if the two soldiers sent to point C are stopped before they can successfully capture it, they still distract HQ-DoG from attacking points A and B, something Static Strategy (5,5,0) does not do at all.

Despite the fact that it loses most of its games, HQ-DoG is able to learn competent strategies to handle different circumstances. Table 4 shows the portion of the captain’s Q-table for the state where HQ-DoG has captured points B and C and has lost point A. The Q-values indicate that against Static Strategy (4,4,2) the best action to take in this state is devote the entire team to the defense of point B and ignore all other points. This is smart because even though Static Strategy (4,4,2) will attack point C, its main focus is still to maintain control of points A and B, so that's where the majority of its force will be placed. The Q-value for this action is 5.18675. In the event that point C is lost while executing this action, Table 5 shows that HQ-DoG learns it is then best to attack point C with half the team in hopes of recapturing the point and continue to defend point B with the remaining half of the team. The Q-value for this action of simultaneous attack and defense has a Q-value of 4.9106. So even if HQ-DoG does not have the upper hand in a match, it is still able to learn a good strategy because learning takes place on a case by case basis of the current state.

A game can be thought of as a sequence of states, some of which HQ-DoG has the advantage, some which it doesn't. A win or loss is the final product of this sequence of states, but a win doesn't mean all the states in the sequence were good, neither does a loss mean all the states that led to that result were unfavorable. Even if HQ-DoG loses a match, there is still the chance that it learned shrewd strategy sometime during the game. Also there is the fact that HQ-DoG does not reward or punish itself based on whether it won or lost a match. All utility is derived solely on the merit of the current state.

Table 4 - Captain Q-table vs. (4,4,2). Lost point A. Captured points B & C

State	Action	Q-Value	Visits
Lost,Captured,Captured	Attack,Defend,Defend	3.91322	160
Lost,Captured,Captured	Attack,Defend,Ignore	4.23564	345
Lost,Captured,Captured	Attack,Ignore,Defend	3.13522	101
Lost,Captured,Captured	Attack,Ignore,Ignore	3.71065	484
Lost,Captured,Captured	Ignore,Defend,Defend	3.86257	202
Lost,Captured,Captured	Ignore,Defend,Ignore	5.18675	1837
Lost,Captured,Captured	Ignore,Ignore,Attack	3.93011	1198

Table 5 – Captain Q-table vs. (4,4,2). Lost points A & B. Captured B

State	Action	Q-Value	Visits
Lost,Captured,Lost	Attack,Defend,Defend	1.71168	504
Lost,Captured,Lost	Attack,Defend,Ignore	2.30322	219
Lost,Captured,Lost	Attack,Ignore,Attack	1.90236	76
Lost,Captured,Lost	Attack,Ignore,Ignore	0.82281	53
Lost,Captured,Lost	Ignore,Defend,Attack	4.9106	321
Lost,Captured,Lost	Ignore,Defend,Ignore	1.39398	24
Lost,Captured,Lost	Ignore,Ignore,Attack	1.83775	41

5.2.2.3 HQ-DoG vs. Static Strategy (3,4,3)

Figure 24 shows the win/loss chart between HQ-DoG and the Static Strategy (3,4,3). HQ-DoG fairs better in this experiment compared to it competition against the Static Strategy (4,4,2), winning 46.6% of the five hundred matches. Figure 25 displays their score difference, which has an average of -2.498. This static strategy is the most aggressive because it tries to capture all domination points at the same time and spread its forces as evenly over the map as possible. It does not ignore point C, which is why it performs better than Static Strategy (5,5,0). The problem is that in focusing on all targets equally, it spreads its forces thin. This is why HQ-DoG does better against it than against Static Strategy (4,4,2), which focuses on points A and B.

Again, even though HQ-DoG loses the majority of the matches in the experiments, that doesn’t preclude it from learning good strategy. Table 6 shows that against Static Strategy (3,4,3), HQ-DoG learns to attack point B at the beginning of a game with the entire team to ensure it gains early control of the middle of the map. The Q-value for this action is 3.93379. Table 7 shows that after point B is captured, HQ-DoG goes back to capture point C while still defending point B. This action has a Q-value of 5.39847. This can be smart because even though capturing C first would give HQ-DoG points sooner, those faster points might cost it possession of point B. If the team devotes all its forces to capturing point B first, it is likely to kill the enemy soldiers that are also trying to capture point B. Those killed enemy soldiers will respawn back at point A, giving HQ-DoG the opportunity to divert soldiers back to capture point C.

Sometimes HQ-DoG captures all the domination points when playing against Static Strategy (3,4,3). In these cases, Table 8 shows that the algorithm learns that it is best to try to defend all points for as long as it can which makes sense because at that point it will earn three points every five seconds and the enemy will earn none. The Q-value for this action is 3.585859. Table 9 shows that when the defensive lieutenant is given the task of defending all points and it is observed that there are enemy soldiers all over the map, it tries to spread its soldiers out as evenly as possible. This gives HQ-DoG the best chance of defending the three points against enemy attack. The Q-value for this action is 2.09847. This shows that both the captain and lieutenants are able to learn good strategy when facing a competitive opponent.

Table 6 - Captain Q-table vs. (3,4,3). All points neutral

State	Action	Q-value	Visits
Neutral,Neutral,Neutral	Attack,Attack,Attack	3.585859	101
Neutral,Neutral,Neutral	Attack,Attack,Ignore	2.437947	116
Neutral,Neutral,Neutral	Attack,Ignore,Attack	2.53359	6
Neutral,Neutral,Neutral	Attack,Ignore,Ignore	-0.368639	6
Neutral,Neutral,Neutral	Ignore,Attack,Attack	2.068462	6
Neutral,Neutral,Neutral	Ignore,Attack,Ignore	3.933379	259
Neutral,Neutral,Neutral	Ignore,Ignore,Attack	0.773485	6

Table 7 - Captain Q-table vs. (3,4,3). Lost point A. Captured point B. Point C is neutral

State	Action	Q-value	Visits
Lost,Captured,Neutral	Attack,Defend,Defend	3.50264	206
Lost,Captured,Neutral	Attack,Defend,Ignore	2.91887	8
Lost,Captured,Neutral	Attack,Ignore,Attack	2.84923	4
Lost,Captured,Neutral	Attack,Ignore,Ignore	0.32096	2
Lost,Captured,Neutral	Ignore,Defend,Attack	5.39847	113
Lost,Captured,Neutral	Ignore,Defend,Ignore	1.36745	3
Lost,Captured,Neutral	Ignore,Ignore,Attack	2.28632	5

Table 8 - Captain Q-table vs. (3,4,3). All points captured

State	Action	Q-value	Visits
Captured,Captured,Captured	Defend,Defend,Defend	2.26174	36
Captured,Captured,Captured	Defend,Defend,Ignore	0.46074	1
Captured,Captured,Captured	Defend,Ignore,Defend	0.83213	1
Captured,Captured,Captured	Defend,Ignore,Ignore	1.37616	2
Captured,Captured,Captured	Ignore,Defend,Defend	1.00711	1
Captured,Captured,Captured	Ignore,Defend,Ignore	0.37641	1
Captured,Captured,Captured	Ignore,Ignore,Defend	0.63632	1

Table 9 - Defense Lt.'s Q-table vs. Static Strategy (3,4,3).
State: Defend all points. Enemy seen near all targets.

Assignment	Observation	Force	Distribution	Q-Value	Visits
A B C	True,True,True	100%	[3,4,3]	2.09847	24
A B C	True,True,True	100%	[2,4,4]	0.44569	1
A B C	True,True,True	100%	[4,2,4]	0	0
A B C	True,True,True	100%	[4,4,2]	0	0

5.2.2.4 HQ-DoG vs. Static Strategy (6,2,2) and (8,1,1)

Table 10 displays how HQ-DoG compares against all the Static Strategies. It shows that the Static Strategies (6,2,2) and (8,1,1) fared poorly as expected. Even though their strategies employ coordination, they ineffectively deploy their soldiers across the domination points. As stated in Section 5.1.2, these two strategies are too defensive around point A and HQ-DoG capitalizes on this winning 98.2% of its matches against Strategy (6,2,2) and 99.4% of its matches against Strategy (8,1,1). This shows that coordination alone is not enough to be competitive in the domination gametype and defeat HQ-DoG. An opponent static strategy must have both coordination and an effective deployment of its soldiers across the map.

Table 10 - HQ-DoG vs. Static Strategies (6,2,2) and (8,1,1)

Strategy	Avg. Score Diff	Win Ratio
(5,5,0)	8.656	54.8%
(4,4,2)	-7.138	41.4%
(3,4,3)	-2.498	46.6%
(6,2,2)	39.264	98.2%
(8,1,1)	47.78	99.4%

5.2.3 HQ-DoG vs. Extended Experiment Static Strategy (4,4,2)

The previous experiments allow HQ-DoG to learn its strategy within five hundred matches. There is the question of whether performance against a particular strategy can be improved if the algorithm is given more time to learn. There is also the question as to how the currently learned Q-table will perform if its learning is frozen but it is allowed to continue to compete. To answer these questions an extended experiment is conducted with the Static Strategy that performed the best against HQ-DoG. Static Strategy (4,4,2) won the most games out of all the Static Strategies, so it is the one chosen for an extended experiment of 1000 games. Two separate trials are run in the extended experiment; they both use the same captain and lieutenant Q-tables acquired in the first 500-game experiment for Static Strategy (4,4,2). One trial disables learning and plays against Static Strategy (4,4,2) for another 500 games with the frozen Q-tables. The other trial plays another 500 matches but allows the learning process to continue.

This extended experiment compares its trials against the original experiment for Static Strategy (4,4,2) to determine if HQ-DoG can improve its win/loss ratio given more time to compete with a developed Q-table. The extended experiment also compares its trials directly to each other to discover if there is an advantage to be had by allowing the algorithm to continue to learn as it competes versus locking the algorithm and just utilizing what it has learned up to the current point. Table 11 shows the three different runs of HQ-DoG against different versions of Static Strategy (4,4,2). It shows that after freezing the algorithm’s learned Q-tables and running for another 500 matches, HQ-DoG is able to increase the percentage of games won from 41.4% to 52.2%, over half the games in the 1000 matches. This means that HQ-DoG won 63% of the 500 games in the second half of the extended experiment while competing with the Q-tables it had developed during the first 500 games. Table 11 also shows that when learning is allowed to continue, HQ-DoG wins 55.5% of the 1000 matches, meaning in that trial HQ-DoG wins 69.6% of the last 500 matches. It indicates that HQ-DoG is able to perform better after it is given the time to develop a policy for its Q-tables and that continued development improves its performance.

Table 12 shows the comparison between HQ-DoG with and without continued learning against Static Strategy (4,4,2) for the last 500 games of each trial of the extended experiment. It shows that allowing HQ-DoG to continue to learn after it has developed a policy improves performance over competing with frozen Q-tables. The trial that uses continued learning has a higher average lead and a higher win percentage than the trial that locks the learning process.

Table 11 - HQ-DoG vs. Static Strategy (4,4,2) 1000 Games

Strategy	# of Matches	Avg. Score Diff.	HQ-DoG Win %
(4,4,2)	500	-7.138	41.4%
Extended (4,4,2) w/o Learning	1000	0.312	52.2%
Extended (4,4,2) w/ Learning	1000	2.630	55.5%

Table 12 - HQ-DoG vs. Static Strategy (4,4,2) Second 500 Games

Trial	Avg. Score Diff	HQ-DoG Win%
Extended (4,4,2) w/o Learning	7.762	63.0%
Extended (4,4,2) w/ Learning	12.398	69.6%

5.2.4 HQ-DoG vs. Random Strategy

Out of all the experiments with the static strategies, HQ-DoG has the most difficulty contending with Strategies (5,5,0), (4,4,2), and (3,4,3). Because of this the Random Strategy chooses randomly from these three static strategies. Figure 26 displays the win/loss chart of HQ-DoG against the Random Strategy. It shows that it is more difficult for HQ-DoG to learn a policy against the Random Strategy compared to most of the other strategies. In this experiment, it wins only 45.8% of the matches. Figure 27 shows the difference in scores against Random Strategy, the average of which is -2.774, one of the worst of all. By choosing a strategy at random, the Random Strategy makes it harder for HQ-DoG to learn a competitive policy against any one of the chosen strategies in particular. HQ-DoG must adapt to a new strategy at the start of each game which is comparable to erasing its Q-table and starting from scratch. This shows changing strategies each match and not doing the same thing repeatedly, is an effective line of attack against HQ-DoG. This would give a team of players more entertainment value because they would have to constantly change their gameplay if they wanted to win matches against HQ-DoG.

5.3 Discussion

The results of these experiments show that HQ-DoG is capable of learning a competitive policy against different strategies for the domination gametype in first person shooter games. It is far superior to the Greedy Strategy and Static Strategies (8,1,1) and (6,2,2), winning over 94% of the matches in all cases. It is able to win over half the matches it has against Static Strategy (5,5,0), which is competitive given that Static Strategy (5,5,0) uses one of the best strategies there is for playing domination and HQ-DoG must learn a policy from scratch. Against a more difficult enemy like Static Strategy (4,4,2) HQ-DoG is able to win approximately 40% of the matches. However, HQ-DoG shows that it can compete better against this same strategy if it plays again with the policy it has just learned. The results show that after a policy is developed, HQ-DoG wins over 60% of the games in another 500-match experiment with Static Strategy (4,4,2). When the developed policy is allowed to continue learning as it competes, it wins almost 70% of the games in another 500-match experiment with Static Strategy (4,4,2). This shows that learning does take place and that it improves performance of HQ-DoG over time. By examining its Q-tables, HQ-DoG shows that it is capable of learning the appropriate decision in different circumstances for both offense and defense. These decisions include where to send soldiers as well as how many to send. The results show that one of the best ways to compete against HQ-DoG is to change strategies and not stick to one plan. In the last experiment the Random Strategy wins approximately 55% of the matches. This shows that if a team wishes to win consistently against HQ-DoG the team must change its strategy so that HQ-DoG can not learn to counter it.

It is important to note that the randomness found in the Greedy Strategy is different from that found with the Random Strategy. The Greedy Strategy is unpredictable in nature mostly because there is neither communication nor any coordination among soldiers. All the soldiers act independently of one another so any cooperation is emergent. This type of randomness is very ineffective against HQ-DoG. Alternatively, the Random Strategy is unpredictable only in what strategy it will employ each game match. Once it has chosen a strategy it sticks with it for the duration of the match, and to do so it must explicitly coordinate its soldiers. Soldiers are split off into groups and assigned a domination point. Even though the soldiers in a group don't communicate with each other directly their cooperation is by design because they were all deliberately sent to the same spot. This purposeful use of randomness and coordination is what gives the Random Strategy its edge.[2] It shows that if a team wants to be successful against HQ-DoG it must change its strategies over time; otherwise the algorithm will eventually learn a policy that is able to counteract it and given enough time begin to defeat it.

It is also important to note that the purpose of HQ-DoG is not to learn a policy that wins every match. As stated before, even if the optimal decision is always made, winning the match is not guaranteed because the environment is stochastic. Part of victory is based on the position of the enemy and how well a team's soldiers fair against the opposition in combat. Instead the purpose of HQ-DoG is to find a policy that is competitive against the opposing team's strategy to provide a fun and challenging experience. Fast convergence to an optimal policy is not the primary concern. The fact that the algorithm adapts to gameplay and makes the opposing team change its strategies is what's most important. By doing this, HQ-DoG gives the opposing team a different experience from one game to the next which makes the game interesting and entertaining.

[2] The Random Strategy must choose among competitive strategies like (4,4,2) and (5,5,0) for an advantage though. Choosing among poor performing strategies like (8,1,1) will still result in mostly losses despite the random factor.

CONCLUSION

This paper has presented HQ-DoG, an algorithm that employs hierarchical Q-learning to direct the actions of a team of ten soldiers in the domination gametype for a first person shooter simulation. Whereas previous work employs conventional RL in this area for control of up to only five agents, this paper contends that controlling larger groups of agents with these methods grows increasingly difficult because the state-action space of the domain grows exponentially with the number of soldiers on a team. This is compounded by the fact that the tasks of learning which targets to attack or defend as well as how many soldiers to commit to each target are learned simultaneously in the same Q-table. For these reasons, HQ-DoG proposes a hierarchical architecture consisting of three separate Q-tables that represent a captain and two subordinate lieutenants. The captain learns a competitive policy for which points to attack and defend given the state of the environment (which flags it does and doesn't own). The lieutenants are responsible for offense and defense and, given the orders they receive from the captain, learn how to distribute the team's forces across the targets. This divides the task of learning where and how to deploy the soldiers into two easier to solve subtasks. HQ-DoG reduces the state-space and abstracts the information used by the Q-tables compared to other methods and in doing so proves it can learn a winning policy for controlling a large number of soldiers.

HQ-DoG is able to scale well to a domain with a larger team size. The state-action space of the captain's Q-table remains the same regardless of team size because its state is based on the 27 possible combinations of the three domination points which do not change. Only a lieutenant's state-action space has the potential to grow with a different team size, but the designer has some control over this growth. Lieutenants choose among percentages for different soldier distributions, and these percentages as well as the number of possible distributions can change at the designer's discretion. The designers can have the lieutenants choose from as many distributions as he or she feels is adequate. Given N number of soldiers per team, it is not necessary to make every mathematically possible distribution across the three points available because not every distribution deploys the soldiers effectively across the map.

There is further research that can be done with HQ-DoG. One direction is to implement the algorithm across a network. At the moment the captain, lieutenants, and soldiers are tied together, but this is not a requirement of the algorithm, it is only a restriction of the development environment it is deployed in. The components of the algorithm can be separated as long as they are able to communicate with each other. It is of interest to investigate how HQ-DoG can perform in a client-server capacity where embodied soldiers on the team execute actions in the simulation environment at one location, but the disembodied AI sits somewhere else, passing instructions from a different location. There are issues to explore like the implications of how long it takes for data to travel across the network and what happens if the network fails to deliver information in a timely manner or at all.

Another direction of research is to investigate how HQ-DoG behaves with a different utility function, particularly one that doesn't explicitly take the enemy into consideration, like what is found in RL-DOT. HQ-DoG subtracts the number of enemy-owned points from the number of points it owns to directly calculate utility from the environment. This means there is a different utility for when HQ-DoG owns one point and the enemy owns none as opposed to when they both own one. Alternatively RL-DOT, assigns utility based only on the number of points that it owns, so it would give the same reward under both circumstances. HQ-DoG uses a wider range for utility (seven values compared to four) but it is uncertain at this point whether that leads to better performance. A wider range of utility values allows HQ-DoG to detect more circumstances and score them differently, but this might be unnecessary. Further study would be required to determine if a change in utility function would result in a more or less competitive policy.

REFERENCES

Albus, J. S., Anthony, J. R., & Roger, N. N. (1981). Theory and Practice of Hierarchical Control. *Proc 23rd IEEE Computer Society International Conference*, (pp. 19-39).

Atkin, M. S., Westbrook, D. L., & Cohen, P. R. (1999). Capture the Flag: Military Simulation Meets Computer Games. *Symposium on Artificial Intelligence and Computer Games* (pp. 1-5). Menlow Park, CA: AAAI Press.

Auslander, B., Lee-Urban, S., Hogg, C., & Munoz-Avila, H. (2008). Recognizing the Enemy: Combining Reinforcement Learning with Strategy Selection using Case-Based Reasoning. *In Proceedings of the Ninth European Conference on Case-Based Reasoning (ECCBR 2008),* (pp. 59-73). Trier, Germany.

Bensaid, N., & Mathieu, P. (1997). A Hybrid and Hierarchical Multi-Agent Architecture Model. *PAAM*, (pp. 145-155).

Bourg, D., & Seeman, G. (2004). *AI For Game Developers.* Beijing: O'Reiily Media Inc.

Champard, A. J. (2007, September 6). *Understanding Behavior Trees*. Retrieved June 4, 2012, from AI Game Dev: aigamedev.com/open/article/bt-overview

Champard, A. J. (2012a, Febraury 9). *Trends and Highlights in Game AI for 2011*. Retrieved June 3, 2012, from AI Game Dev: aigamedev.com/insider/discussion/2011-trends-highlights/

Champard, A. J. (2012b, February 26). *Understanding the Second-Generation of Behavior Trees*. Retrieved June 3, 2012, from AI Game Dev: aigamedev.com/insider/tutorial/second-generation-bt/

Dignum, F., Westra, J., van Douesburg, W. A., & Harbers, M. (2009). Games and Agents: Designing Intelligent Gameplay. *International Journal of Computer Games Technology*. Hindawi Publishing Corporation.

Dignum, V., Dignum, F., & Sonenberg, L. (2004). Towards Dynamic Reorganization of Agent Societies. *CEAS: Workshop on Coordination in Emergent Agent Sociesties at ECAI*, (pp. 22-27). Valencia: Spain.

Ferber, J., & Gutknecht, O. (1998). A Meta-Model for the Analysis and Design of Organization in Multi-Agent Systems. *Third International Conference of Multi-Agent Systems,* (pp. 128-135). Paris, France.

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *International joint Conferences on Artificial Intelligence*, (pp.189-208).

Ghavamzadeh, M., Mahadevan, S., & Makar, R. (2006). Hierarchical Multi-Agent Reinforcement Learning. *Autonomous Agents Multi-Agent System*, (pp. 197-229).

Ghijsen, M., Jansweijer, W. N., & Wielinga, B. J. (2010). Adaptive Hierarchical Multi-Agents Organizations. *Interactive Collaborative information systems*, (pp. 375-400).

Glaser, N., & Morginot, P. (1997). The Reorganization of Societies of Autonomous Agents. *Eighth European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, (pp. 98-111).

Gosavi, A. (2009). *A Tutorial for Reinforcement Learning.* Rolla, MO: Department of Enginerring Management and System Engineering Missouri Uinversity of Science and Technology.

Hoang, H., Lee-Urban, S., & Munoz-Avila, H. (2005). Hierarchical Plan Representations for Encoding Strategic Game AI. *In Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05).* AAAI Press.

Hogg, C., Lee-Urban, S., Munoz-Avila, H., Auslander, B., & Smith, M. (2011). Game AI for Domination Games. *Artificial Intelligence for Computer Games*, (pp. 83-102).

Kraus, S. (1997). Negotiation and Cooperation in Multi-Agent Environments. *Artificial Intelligence*, (pp. 79-98).

Lees, M., Logan, B., & Theodoropoulos, G. K. (2006). Agents, Games, and HLA. *Simulation Modelling Practice and Theory*, (pp. 752-767).

MacDonald, K. (2011, November 11). *Modern Warfare 3 has Biggest Launch of Anything Ever*. Retrieved November 14, 2011, from www.ign.com: http://xbox360.ign.com/articles/121/1212246p1.html

McPartland, M., & Gallagher, M. (2008a). Learning to be a Bot: Reinforcement Learning in Shooter Games. *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, (pp. 78-83). Stanford, California.

McPartland, M., & Gallagher, M. (2008b). Creating a Multi-Purpose First Person Shooter Bot with Reinforcement Learning. *In Proceedings of IEEE Symposium on Comnputation Intelligence ang Games (CIG '09)*, (pp. 143-150).

Mehta, N., Tadepalli, P., & Fern, A. (2009). *MASH: A Scalable Multi-Agent Framework for Hierarchical Reinforcement Learning.* AAAI Press.

Miikkulainien, R. (2006). Creating Intelligent Agents in Games. *The Bridge*, (pp. 5-13).

Mikkulanien, R., Bryant, B. D., Cornelius, R., Karpov, I. V., Stanley, K. O., & Yong, C. H. (2006). Computational Intelligence in Games. *Computational Intelligence: Principles and Practice*, (pp. 155-191).

Mohd Shukri, S. R., & Mohd Shaukhi, M. K. (2008). A Study on Multi-Agent Behavior in a Soccer Game Domain. *World Academy of Science, Engineering and Technology*, (pp. 308-312).

Nau, D., Cao, Y., Lotem, A., & Munoz-Avila, H. (1999). SHOP: Simple Hierarchical Ordered Planner. *Proceedings of the 16th International Joint Conference on Artificial Intelligence.* San Francisco, CA: Morgan Kaufmann Publicers Inc.

O'Connor, A. (2009, November 18). *Activision Boasts Modern Warfare 2 Sales Figures, Broken Records*. Retrieved Decebmer 12, 2009, from Shack News: http://www.shacknews.com/article/61298/activision-boasts-modern-warfare-2

Orkin, J. (2006). Three States and a Plan: The AI of F.E.A.R. *In Proceedings of Game Developer's Conference.*

Patel, P. G., Carver, N., & Rahimi, S. (2011). Tning Computer Gaming Agents using Q-Learning. *Proceeding of the Federated Conference on Computer Science and Information Systems*, (pp. 581-588).

Rao, A. S. (1996). AgentSpeak(L): BDI Agents speak out in a logical computational language". *Proceedings of the 7th Workshop on Modeling Autonomous Agents in a Multi-Agent World (MAAMAW'96)* (pp. 42-45). London: Spring-Verlag.

Rao, A. S., & Georgegg, M. P. (1995). BDI Agents: From Theory to Practice. *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS-95)*, (pp. 312-319). San Francisco, USA.

Routier, J. C., Mathieu, P., & Secq, Y. (2001). Dynamic Skills Learning: A support to Agent Evolution. *AISB Symposium on Adaptive Agents and Multi-Agents Systems*, (pp. 25-32).

Smith, M., Lee-Urban, S., & Munoz-Avila, H. (2007). RETALIATE: Learning Winning Policies in First-Person Shooter Games. *In Proceddings of the Seventeenth Innovative Applications of Artificial Intelligence Conference (IAAI-07).* AAAI Press.

Straatman, R., van der Sterren, W., & Beij, A. (2005). Killzone's AI: dynamic procedural combat tactics. *In Proceedings of Game Developer's Conference.*

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence*, (pp. 181-211).

van Eck, N. J., & Wezel, M. (2008). Application of Reinforcement Learning to the Game of Othello. *Computers & Operations Research*, (pp. 1999-2017).

van Oijen, J., van Doesburg, W., & Dignum, F. (2011). Goal-Based Communication Using BDI Agents as Virtual Humans in Training: An Ontology Driven Dialogue System. *Agents for Games and Simulations II*, (pp. 38-52).

Walton, B. (2009, November 11). *VG Chartz*. Retrieved December 12, 2009, from Modern Warfare 2 sells 7 million copies on day one: http://www.vgchartz.com/article/5826/modern-warfare-2-sells-7-million-copies-on-day-one/

Wang, H., Gao, Y., & Chen, X. (2010, March). RL-DOT: A Reinforcement Learning NPC Team for Playing Domination Games. *IEEE Transactions on Computational intelligence and AI in Games*, (pp. 17-26).

Wooldridge, M., Jennings, N. R., & Kinny, D. (1999). A Methodology for Agent-Oriented Analysis and Design. *International Conference of Autonomous Agents*, (pp. 69-76). Seattle, Washington.

Appendix A – Captain States and Actions

A.1 Captain States

A	B	C
Lost	Lost	Lost
Lost	Lost	Captured
Lost	Lost	Neutral
Lost	Captured	Lost
Lost	Captured	Captured
Lost	Captured	Neutral
Lost	Neutral	Lost
Lost	Neutral	Captured
Lost	Neutral	Neutral
Captured	Lost	Lost
Captured	Lost	Captured
Captured	Lost	Neutral
Captured	Captured	Lost
Captured	Captured	Captured
Captured	Captured	Neutral
Captured	Neutral	Lost
Captured	Neutral	Captured
Captured	Neutral	Neutral
Captured	Lost	Lost
Neutral	Lost	Captured
Neutral	Lost	Neutral
Neutral	Captured	Lost
Neutral	Captured	Captured
Neutral	Captured	Neutral
Neutral	Neutral	Lost
Neutral	Neutral	Captured
Neutral	Neutral	Neutral

A.2 Captain Actions and Associated IDs

Action ID	A	B	C
1	Attack	Attack	Attack
2	Attack	Attack	Defend
3	Attack	Attack	Ignore
4	Attack	Defend	Attack
5	Attack	Defend	Defend
6	Attack	Defend	Ignore
7	Attack	Ignore	Attack
8	Attack	Ignore	Defend
9	Attack	Ignore	Ignore
10	Defend	Attack	Attack
11	Defend	Attack	Defend
12	Defend	Attack	Ignore
13	Defend	Defend	Attack
14	Defend	Defend	Defend
15	Defend	Defend	Ignore
16	Defend	Ignore	Attack
17	Defend	Ignore	Defend
18	Defend	Ignore	Ignore
19	Ignore	Attack	Attack
20	Ignore	Attack	Defend
21	Ignore	Attack	Ignore
22	Ignore	Defend	Attack
23	Ignore	Defend	Defend
24	Ignore	Defend	Ignore
25	Ignore	Ignore	Attack
26	Ignore	Ignore	Defend

A.3 Captain State-Action Space

A	B	C	Valid Action IDs
Lost	Lost	Lost	1,3,7,9,19,21,25
Lost	Lost	Captured	2,3,8,9,20,21,26
Lost	Lost	Neutral	1,3,7,9,19,21,25
Lost	Captured	Lost	4,6,7,9,22,2,25
Lost	Captured	Captured	5,6,8,9,23,24,26
Lost	Captured	Neutral	4,6,7,9,22,24,25
Lost	Neutral	Lost	1,3,7,9,19,21,25
Lost	Neutral	Captured	2,3,8,9,20,21,26
Lost	Neutral	Neutral	1,3,7,9,19,21,25
Captured	Lost	Lost	10,12,16,18,19,21,25
Captured	Lost	Captured	11,12,19,18,20,21,26
Captured	Lost	Neutral	10,12,16,18,19,21,25
Captured	Captured	Lost	13,15,16,18,22,24,25
Captured	Captured	Captured	14,15,17,18,23,24,26
Captured	Captured	Neutral	13,15,16,18,22,24,25
Captured	Neutral	Lost	10,12,16,18,19,21,25
Captured	Neutral	Captured	11,12,17,18,20,21,26
Captured	Neutral	Neutral	10,12,16,18,19,21,25
Neutral	Lost	Lost	1,3,7,9,19,21,25
Neutral	Lost	Captured	2,3,8,9,20,21,26
Neutral	Lost	Neutral	1,3,7,9,19,21,25
Neutral	Captured	Lost	4,6,7,9,22,24,25
Neutral	Captured	Captured	5,6,8,9,13,24,26
Neutral	Captured	Neutral	4,6,7,9,22,24,25
Neutral	Neutral	Lost	1,3,7,9,19,21,25
Neutral	Neutral	Captured	2,3,8,9,20,21,26
Neutral	Neutral	Neutral	1,3,7,9,19,21,25

Appendix B – Lieutenant State and Actions

B.1 Derived State Space of a Lieutenant

A	B	C	% Attackers	% Defenders
Attack	Attack	Attack	100%	0%
Attack	Attack	Defend	66%	33%
Attack	Attack	Ignore	100%	0%
Attack	Defend	Attack	66%	33%
Attack	Defend	Defend	33%	66%
Attack	Defend	Ignore	50%	50%
Attack	Ignore	Attack	100%	0%
Attack	Ignore	Defend	50%	50%
Attack	Ignore	Ignore	100%	0%
Defend	Attack	Attack	66%	33%
Defend	Attack	Defend	33%	66%
Defend	Attack	Ignore	50%	50%
Defend	Defend	Attack	33%	66%
Defend	Defend	Defend	0%	100%
Defend	Defend	Ignore	0%	100%
Defend	Ignore	Attack	50%	50%
Defend	Ignore	Defend	0%	100%
Defend	Ignore	Ignore	0%	100%
Ignore	Attack	Attack	100%	0%
Ignore	Attack	Defend	50%	50%
Ignore	Attack	Ignore	100%	0%
Ignore	Defend	Attack	50%	50%
Ignore	Defend	Defend	0%	100%
Ignore	Defend	Ignore	0%	100%
Ignore	Ignore	Attack	100%	0%
Ignore	Ignore	Defend	0%	100%

B.2 Translation between team percentage of forces and actual size

Percentage of Forces	Actual Size of Force
0%	0
33%	3
50%	5
66%	7
100%	10

B.3 Action Space of Lieutenants

Action IDs	Distribution of Allotted Forces Across Targets	Number of Targets Assigned
1	(33%, 33%, 33%)	3 Targets
2	(20%, 40%, 40%)	3 Targets
3	(40%, 20%, 40%)	3 Targets
4	(40%, 40%, 20%)	3 Targets
5	(33%, 66%)	2 Targets
6	(50%, 50%)	2 Targets
7	(66%, 33%)	2 Targets
8	100%	1 Target

B.4 Break down of lieutenant distribution across assigned targets given different percentages of the entire team

Action ID	1st	2nd	3rd
Three Targets, Ten Players (100%)			
1	3	4	3
2	2	4	4
3	4	2	4
4	4	4	2
Two Targets, Ten Players (100%)			
5	3	7	
6	5	5	
7	7	3	
Two Targets, Seven Players (66%)			
5	2	5	
6	3	4	
7	5	2	
One Target, Ten Players (100%)			
8	10		
One Target, Five Players (50%)			
8	5		
One Target, Three Players (33%)			
8	3		

B.5 Observation Model Dependent on Number of Targets Assigned

1st	2nd	3rd
True	True	True
True	True	False
True	False	True
True	False	False
False	True	True
False	True	False
False	False	True
False	False	False

1st	2nd
True	True
True	False
False	True
False	False

1st
True
False

B.6 Entire Lieutenant State-Action Space

Assignment			Observation			Force	Actions
A	B	C	TRUE	TRUE	TRUE	100%	1,2,3,4
A	B	C	TRUE	TRUE	FALSE	100%	1,2,3,4
A	B	C	TRUE	FALSE	TRUE	100%	1,2,3,4
A	B	C	TRUE	FALSE	FALSE	100%	1,2,3,4
A	B	C	FALSE	TRUE	TRUE	100%	1,2,3,4
A	B	C	FALSE	TRUE	FALSE	100%	1,2,3,4
A	B	C	FALSE	FALSE	TRUE	100%	1,2,3,4
A	B	C	FALSE	FALSE	FALSE	100%	1,2,3,4
A	B		TRUE	TRUE		66%	5,6,7
A	B		TRUE	FALSE		66%	5,6,7
A	B		FALSE	TRUE		66%	5,6,7
A	B		FALSE	FALSE		66%	5,6,7
A	B		TRUE	TRUE		100%	5,6,7
A	B		TRUE	FALSE		100%	5,6,7
A	B		FALSE	TRUE		100%	5,6,7
A	B		FALSE	FALSE		100%	5,6,7
A		C	TRUE		TRUE	66%	5,6,7
A		C	TRUE		FALSE	66%	5,6,7
A		C	FALSE		TRUE	66%	5,6,7
A		C	FALSE		FALSE	66%	5,6,7
A		C	TRUE		TRUE	100%	5,6,7
A		C	TRUE		FALSE	100%	5,6,7
A		C	FALSE		TRUE	100%	5,6,7
A		C	FALSE		FALSE	100%	5,6,7

Assignment			Observation			Force	Actions
	B	C		TRUE	TRUE	66%	5,6,7
	B	C		TRUE	FALSE	66%	5,6,7
	B	C		FALSE	TRUE	66%	5,6,7
	B	C		FALSE	FALSE	66%	5,6,7
	B	C		TRUE	TRUE	100%	5,6,7
	B	C		TRUE	FALSE	100%	5,6,7
	B	C		FALSE	TRUE	100%	5,6,7
	B	C		FALSE	FALSE	100%	5,6,7
A			TRUE			33%	8
A			FALSE			33%	8
A			TRUE			50%	8
A			FALSE			50%	8
A			TRUE			100%	8
A			FALSE			100%	8
	B			TRUE		33%	8
	B			FALSE		33%	8
	B			TRUE		50%	8
	B			FALSE		50%	8
	B			TRUE		100%	8
	B			FALSE		100%	8
		C			TRUE	33%	8
		C			FALSE	33%	8
		C			TRUE	50%	8
		C			FALSE	50%	8
		C			TRUE	100%	8
		C			FALSE	100%	8