# Game AI for Domination Games

Chad Hogg and Stephen Lee-Urban and Héctor Muñoz-Avila, Bryan Auslander, and Megan Smith

**Abstract** In this paper we present an overview of several techniques we have studied over the years to build game AI for domination games. Domination is a game style in which teams compete for control of map locations, and has been very popular over the years. Due to the rules of the games, good performance is mostly dependent on overall strategy rather than the skill of individual team members. Hence, this makes domination games an ideal testbed to study game AI.

**Key words:** game AI, case-based reasoning, planning, reinforcement learning

## 1 Introduction

*Domination* is a game style in which teams of players compete to control certain locations on a map called *domination points* within a real-time environment. Specifically, a domination point is controlled by the team whose player last stepped on it. Each second, the teams earn points for each of the domination points that they currently control and have controlled for all of some preceding time window. In addition to moving around on the map, players are able to engage in combat with players from the opposite team when they are nearby. A player who is killed in combat respawns at a point randomly selected from a set of pre-determined *spawn points* on the map. In the meantime, the player who killed them may be able to take advantage of this to gain control of a domination point that the killed player had been defending.

Dept. of Computer Science & Engineering
Lehigh University
19 Memorial Drive West
Bethlehem, PA 18015
e-mail: {cmh204,sml3,hem4,bla204,mev2}@lehigh.edu

Domination games have been used, either exclusively or as an option, in a variety of game genres, including first-person shooters (e.g., Half-Life ®, Call of Duty ®), role-playing games (e.g., World of Warcraft ®), and third-person shooters (e.g., Gears of War 2 ®). In addition, many other games that do not perfectly fit the domination model have similar characteristics. For example, Counter-Strike ®, which is among the most popular multiplayer games ever released, also consists of two teams competing in a real-time environment in which strategic control of certain locations (such as bomb sites and hostage drop-offs) is vital to success.

Although an individual player who is highly skilled in combat gives his team a definite advantage, this is much less true than in other game types where, for example, points are awarded for each kill. Instead, team-based strategy is of high importance in these games. Although no player has control over the actions of his teammates, many of these games have built-in communication systems to allow players to devise and execute specific strategies with their teammates. We suspect that the team-oriented nature of these games is a primary reason for their enduring success, and it also makes them an excellent testbed for AI development.

In addition, domination games can generally be classified as having the following properties: Domination games are *non-deterministic*; success in combat requires both skill and luck, and it is not possible to predict whether or not a player will successfully reach his objective. Domination games are also *adversarial*; two or more teams compete to control the domination points. Finally, domination games are *imperfect information* games; a team only knows the locations of those opponent players that are within the range of view of one of the team's own players. These conditions make domination games a good testbed for evaluating algorithms that integrate planning and execution.

We refer to individual players who are not human-controlled as *bots*. The purpose of our research is not to improve the combat performance of individual bots, and so we use the same Finite State Machine-based bot logic for all of our computer-controlled players. Our interest is in the overarching strategies that teams of bots pursue.

Over the years we have devised several methods that integrate planning and execution for selecting a team's strategy in domination games. Table 1 shows a summary of the three algorithms that we will discuss in this paper: HTNBOTS, RETALIATE, and CBRETALITE. HTNBOTS uses hierarchical task network (HTN) representation techniques to generate new plans [3]. It monitors the current situation in the game; when the circumstances change, it generates new plans on the fly. RETALIATE uses reinforcement learning techniques; it uses a Q-learning algorithm to find policies that represent competent ways to play the game [16]. The third system is CBRETALITE [1]. CBRETALITEis built on top of RETALIATE; it stores and retrieves a library of policies, which are reused by using the reinforcement learning algorithm from RETALIATE.

We have conducted a study that compares these three approaches. In this chapter, we report on the architectures of these three systems and comparisons among the knowledge requirements and performance results of the three systems.

**Table 1** Three systems for playing domination games

| Game AI | Description |
|---|---|
| HTNBOTS | Replanning algorithm. Uses HTN planning techniques to generate plans on-the-fly |
| RETALIATE | Generates policies that adapt to the opponent using reinforcement learning techniques |
| CBRETALITE | Stores and reuses policies generated by RETALIATE |

## 2 DOM: A Generic Domination Game Environment

Our initial experiments with each of our three systems used the commercially available game Unreal Tournament ® as the simulation environment. While this simulator provides a useful API for controlling teams of bots, we found that a number of factors made it difficult to perform large-scale experiments and extract useful data from them.

Instead, we built a game environment, called DOM, which captures the essence of domination games [1]. The basic rules in DOM are the following: Each time a bot on team $t$ passes over a domination point, that point will belong to $t$. Team $t$ receives one point for every five game ticks that it owns a domination point. Teams compete to be the first to earn a predefined number of points. No awards are given for killing an opponent team's bot, which *respawns* immediately in a location selected randomly from a set of map locations, and then continues to play. A location is captured by a team whenever one of its bots moves on top of the location and within the next five game ticks no bot from another team moves on top of that location.

The total number of possible states in the game is at least $O(2 * 10^{34})$, assuming a standard map of 70x70 cells, 4 domination locations, and 3 bots per team [2]. It would be infeasible for our agents to reason in such a complicated world, so we have used an abstraction of states and actions that was first described in the work on RETALIATE [16]. In the abstracted description of the world, the current state consists only of the ownership of each domination point; each point is either owned by one of the teams or is unowned at any point in time. Thus, for an environment with $d$ domination points and $t$ teams, the total number of possible states is $d^{t+1}$. We also abstract away the possible actions of the bots. In this abstraction, each action states only to which domination location each of the bots on a team should go. Thus, for an environment with $d$ domination points and $b$ bots per team, the number of actions available to a team is $b^d$. The details of how a bot moves from one location to another are strictly determined by a shortest-path algorithm.

## 3 The Game AI Systems

We briefly summarize each of the three algorithms we investigated that integrate planning and execution for playing DOM. Each of the algorithms focus on controlling which domination locations team-member bots are sent to. Consequently, the behavior of the individual bots can be pre-determined by a standard FSM. Our algorithms do not make a priori assumptions about what that behavior is, which allows bots to be used as plug-ins. In principle, this allows the design decisions for the team AI to be made independently of the design decisions relating to the control of individual bot behavior. Similarly, by using bots as plug-ins, the game developer can swap different bot types in and out of the game and even use bots developed for single-player non-team modes in multi-player games. For further details of these algorithms please see the references.

### *3.1* HTNBots

HTNBots is a dynamic replanning algorithm that uses hierarchical task network (HTN) planning techniques to generate plans [3, 8]. HTN planning proceeds by decomposing high-level tasks such as *win domination game* into simpler tasks such as *send bot b*1 *to location L*1. There are two kinds of tasks: compound and primitive. Compound tasks, such as *win domination game*, can be further decomposed into subtasks whereas primitive tasks cannot. The primitive tasks denote concrete actions, such as *send bot b*1 *to location L*1. Each level in an HTN adds detail on how to achieve the high-level tasks. The sequencing of the leaves in a fully expanded HTN yields the plan for achieving the high-level tasks.

#### 3.1.1 Planning knowledge in HTNBots

HTN planners require that the planning knowledge be provided in the form of methods and operators. A method encodes how to achieve a compound task and consists of three elements:

- **Head**: The task being achieved, called the *head* of the method
- **Preconditions**: The set of preconditions indicating the conditions that must be fulfilled for the method to be applicable, and
- **Subtasks**: The subtasks needed to achieve the head.

Table 2 shows an example of a method (*?*<*string*> indicates that <*string*> is a variable). The task that this method achieves is that team $T$ gains control of locations ?$L1$ and ?$L2$. This method is applicable when the variables ?$L1$ and ?$L2$ refer to domination locations and the variables ?*bot*1, ?*bot*2, and ?*bot*3 refer to distinct bots on team $T$. The method accomplishes its head by ordering one of the bots to go to

location ?*L*1, another to go to location ?*L*2, and the third to patrol between those two locations.

**Table 2** Example method and operator in HTNBOTS

| Method | Operator |
|---|---|
| Head: Control2Locations(*T*, ?*L*1, ?*L*2)<br>Preconditions:<br>domLocation(?*L*1)<br>domLocation(?*L*2)<br>teamMember(?*bot*1, *T*)<br>teamMember(?*bot*2, *T*)<br>teamMember(?*bot*3, *T*)<br>different(?*bot*1, ?*bot*2, ?*bot*3)<br>Subtasks:<br>sendBot(?*bot*1, ?*L*1)<br>sendBot(?*bot*2, ?*L*2)<br>patrol(?*bot*3, ?*L*1, ?*L*2) | Head: sendBot(*b*, ?*LD*)<br>Preconditions:<br>botLocation(*b*, ?*LC*)<br>Effects:<br>¬ botLocation(?*b*, ?*LC*)<br>botLocation(?*b*, ?*LD*) |

Operators define valid actions in the domain. An operator consists of:

- **Head**: The primitive task that the operator accomplishes.
- **Preconditions**: The conditions that must be true for the operator to be applicable.
- **Effects**: How the current situation changes as a result of applying the operator.

Table 2 also shows an example of an operator. This operator sends a bot *b* to a location ?*LD*. The operator is applicable when bot *b* is at location ?*LC*. After successful execution of the operator, bot *b* will no longer be in location ?*LC*, and will instead be in location ?*LD*. The preconditions and effects of operators allow the planner to construct a plan that will achieve the primitive task *if* the action is executed successfully. Because these plans are executed in a nondeterministic environment, this is not always the case.

### 3.1.2 HTN Planning in HTNBOTS

The following are the steps performed by HTNBOTS to decompose a compound task, *t*:

1. $M \leftarrow$ select all methods whose head matches *t*
2. $m \leftarrow$ select a method from *M* that is applicable
3. decompose *t* with the subtasks of *m*

For selecting an applicable method, *m*, HTNBOTS checks if the preconditions are valid in the current state of the game world. This is accomplished through a communication protocol between HTNBOTS and the game engine. For example, for the method shown in Table 2, each of its six preconditions must be fulfilled in

the game world, which will also result in the variables being instantiated to concrete objects in the game world. If the method is used then it will decompose the task Control2Locations($T$, ?$L1$, ?$L2$) (with proper instantiation of the variables) into the three subtasks indicated in the method.

The three steps above are repeated recursively for each compound task in the subtasks of $m$ until a primitive task is reached. For achieving a primitive task HTNBOTS performs the following steps:

1. $O \leftarrow$ select all operators whose head matches $t$
2. $o \leftarrow$ select an operator from $O$ that is applicable
3. Execute action for operator $o$

One again the communication protocol is used to determine if an operator is applicable by checking if the operator's preconditions are valid in the current state of the game world. The communication protocol is also used to execute the action indicated by the operator. An action is a ground instance of an operator. That is, the operator's variables are instantiated with objects (the particular objects are identified by the game engine when determining if the operator is applicable). The game engine has code for executing actions. For example if the action is sendBot($b33$, $L77$), then the game engine will compute a path from the current location of bot $b33$ to location $L77$. The code for executing the action will also determine what to do in in-game situations such as encountering an opponent.

### 3.1.3 Plan execution

In HTNBOTS each action indicates a concrete activity to be executed by one bot. As a result, HTNBOTS can execute actions in parallel if these are performed by different bots. Once the plan is generated, HTNBOTS will start executing each action in the order indicated by the plan. The following steps are performed for each action $a$ in the plan:

1. check if the bot $b$ assigned for performing $a$ is performing another action; if not then execute $a$
2. if $b$ is performing another action then wait until $b$ is done, and then execute $a$

These steps ensure consistency in the execution of the plan. On the other hand they might unnecessarily delay the executions of other actions (e.g., those actions to be performed by other bots that occur later in the plan). More sophisticated execution control could be implemented (e.g., if a latter action is not dependent on a currently delayed action, it could be executed).

When a plan is executed, HTNBOTS keeps constant track of the preconditions of the method decomposing the top level task (i.e., to win a domination game). If a percentage of these preconditions that are no longer valid is greater than a predefined threshold, a new plan is generated and executed. The rationale is that we only want to change the plan if enough conditions in the game have changed making it necessary to adapt to these changes. Typically methods decomposing the top-level tasks have

preconditions about ownership of the domination locations and the method indicates strategies for dealing with those situations. When domination location ownership changes substantially, it makes sense to immediately generate a new plan to adapt to the new situation. HTN plan generation in HTNBOTS is extremely fast making this process seamlessly.

## *3.2* RETALIATE

We used reinforcement learning (RL) to create RETALIATE which uses Q-learning [15] to acquire winning strategies for games in DOM. Unlike some other forms of learning, RL does not require annotated training examples to learn, nor does RL need an expert to provide feedback to the learner. In RL, interaction with the world is the only way the agent gains information: the agent (1) senses the state of the environment (2) chooses which action to take, (3) performs the action, and (4) receives a (scalar) reward or punishment. Under the RL approach, time is spent crafting the representation of the game state, called the "problem model" - that is, how the various complexities of complete game states are abstracted into a simpler form that RL can use. This is typically significantly easier than manually designing and implementing strategies in complete symbolic representations, such as HTNBOTS . The problem model used by RETALIATE is presented in Section 3.2.2.

In this section, we first briefly describe RL in general, and Q-learning in particular. Next, we present the way in which we modeled the states and actions of DOM in order to increase the efficiency of our application of the Q-learning process. This section is concluded with the RETALIATE algorithm.

### 3.2.1 Reinforcement Learning

Reinforcement Learning is a form of machine learning where an agent or team of agents learns a policy - what action to select in every perceived world state - in a potentially stochastic environment. The goal in RL is to arrive at an optimal policy, which is one that maximizes the rewards received, through a process of trial and error. For an overview of RL in general, see [15].

The purpose of RL algorithms is to find a policy $\pi$ that maximizes the sum of the returned rewards. A policy $\pi$ is a mapping from states to actions indicating for each state $s$, the action $\pi(s)$ that should be chosen. This mapping is calculated by using the rewards received from previous action selections in states already visited. Each state-action value is a representation of the expected future rewards of taking $s$ in $a$, and assumes that policy $\pi$ is used for all subsequent action selections. Rewards are obtained from the environment as a result of the agent's actions, and are measured as $U(s') - U(s)$, the difference between the utilities of the current state $s$ and the next state $s'$ that will be reached after executing an action. In Section 3.2.2 we present our

definition of the utility and reward functions for use of the RETALIATE algorithm in DOM.

In RL, including Q-learning, the choice of which action to take in state $s$, that is $\pi(s)$, involves the use of estimates of the expected value of taking each action in every possible game state. These estimates are derived from the rewards received after taking a selected action in a given state. There exist multiple ways for keeping track of the estimates, and in Q-learning the most straightforward approach is to maintain a "Q-table" that associates with each $(state, action)$ pair the estimated value $Q(s,a)$ of the pair, called the "Q-value". The value of the reward, $R$, is used to perform an update on the Q-table entry $Q(s,a)$ for the previous state $s$ in which the last action $a$ was ordered. The Q-table approach is only feasible when the number of states and actions in the problem model is limited, not only because the table size can become very large otherwise (the size of the table is the number of states multiplied by the number of actions), but also because the amount of learning cycles required to arrive at an accurate estimate of the Q-value grows with the number of state-action pairs.

The update on the Q-table entry $Q(s,a)$ for the previous state $s$ in which the last action $a$ was executed is computed according to the following formula, which is standard for updating the entries in a Q-table in temporal difference learning:

$$Q(s,a) = Q(s,a) + \alpha(R + \gamma * \underset{a'}{\text{argmax}}(Q(s',a') - Q(s,a))) \quad (1)$$

In this computation, the Q-value in the Q-table for the action $a$ that was just taken in state $s$, $Q(s,a)$, is updated. The function argmax returns the value from the Q-table of the best team action that can be performed in the new state, $s'$, which is simply the highest value associated with $s'$ in the table for any $a'$. The value of $\gamma$, which is called the discount-rate parameter, adjusts the importance of future rewards in making current decisions.

In order to safeguard against creating Q-values (and therefore policies) that are stuck in a local optimum, action selection is often performed using an "$\varepsilon$-greedy strategy"; rather than always executing the action of highest estimated value in a given state, when the system is in state $s$, with a probability of $1-\varepsilon$ it selects the action $a$ with the highest $Q(s,a)$ and with a probability $\varepsilon$ it selects a random action. The policy $\pi$ employed by a Q-learning agent in this case is therefore the combination of $\varepsilon$-greedy action selection with a Q-table.

### 3.2.2 Problem Model: Definition of States and Actions

When deciding upon the problem model to use in RL, one must consider the essential features of the problem being addressed. For example, while the amount of ammunition remaining is important for an individual team member, the overall team's strategy might safely ignore this detail. A problem model that takes into consideration too many features of the game state can lead to a learning problem that is very difficult, or impossible, for the system to solve in a reasonable amount of time. Similarly, an overly-simplified problem model leads to a system that does not

play very well, or one that has very limited capabilities. The trick is to model the problem in such a way that learning can happen quickly, while simultaneously being rich enough to support a range of interesting behaviors.

In RETALIATE game states are represented in the problem model as a tuple indicating the owner $O_i$ of the domination location $i$. For instance, if there are three domination locations, the state $(E,F,F)$ describes the state where the first domination location is owned by the enemy and the other two domination locations are owned by our friendly team. Neutral ownership of a domination location is also considered and is represented by an $N$ in the relevant location in the tuple. For three domination locations and two teams, there are twenty-seven unique states of the game, taking into account that domination locations are initially not owned by either team.

The addition of other parameters was considered to increase the information contained in each state. The additional information slowed the RETALIATE learning process considerably, reduced the effectiveness of the RL team, and ultimately was not worth the additional computational cost. In contrast, not only did the simpler definition greatly reduce size of the state space, leading to more rapid learning but, as our result show, it also contained sufficient information to develop a winning policy. The separation of parameters - those used to define team tactics versus those used for individual behavior - is one of the central qualities of RETALIATE.

In RETALIATE states are associated with a set of *team actions*. A team action is defined as a tuple indicating the individual action $A_i$ that bot $i$ takes - for a team of three bots, a team action tuple consists of three *individual actions*. An individual action specifies to which domination location a bot should move. For example, in the team action $(Loc1, Loc2, Loc3)$, the three individual actions send $bot1$ to domination location 1, $bot2$ to domination location 2, and $bot3$ to domination location 3, whereas in $(Loc1, Loc1, Loc1)$, the individual actions send all three bots to domination location 1. If a bot is already in a location that it is told to move to, the action is interpreted as instructing the bot to stay where it is. Individual bot actions are executed in parallel and, for a game with three domination locations and three bots, there are twenty-seven unique team actions because each bot can be sent to three different locations. The Q-table therefore contains $27 \times 27 = 729$ Q-value entries.

Despite the simplicity in the representation of our problem model, it not only proves effective but it actually mimics how human teams play domination games. The most common error of novice players in this kind of game is to fight opponents in locations other than the domination ones; these fights should be avoided because they generally do not contribute to victories in these kinds of games. Part of the reason for that is that, if a player is killed away from a domination location, it will not have a direct affect on ownership and hence will not have an effect on the score. Consequently, it is common for human teams to focus on coordinating to which domination points each team member should go and this is precisely the kind of behavior that our problem model represents.

### 3.2.3 The RETALIATE Algorithm

Algorithm 1 presents pseudocode for the RETALIATE online learning algorithm. RETALIATE is designed to run across multiple game instances so that the policy, and therefore the RETALIATE -controlled team, can adapt continuously to changes in the environment while keeping track of what was learned in previous games. These changes can include changes in our own players (e.g., different type of bot), changes in the opponent team (e.g., changes of tactics), and changes in the game world (e.g., a new map).

**Algorithm 1** RETALIATE$(Q_t)$

**Input**: Q-Table $Q_t$
**Output**: updated Q-table
$\varepsilon$ is 0.1, $\alpha$ is 0.2, $\gamma$ is 1.0, and state $s_{prev}$ is maintained internally
**if** $\mathsf{rand}(0,1) > \varepsilon$ **then** {epsilon greedy selection}
    action $a \leftarrow$ applicable action with max value in Q-table
**else**
    action $a \leftarrow$ random applicable action from Q-table
state $s_{now} \leftarrow \mathsf{Execute}(a)$
reward $R \leftarrow U(s_{now}) - U(s_{prev})$
$Q_t(s_{prev}, a) \leftarrow Q_t(s_{prev}, a) + \alpha(R + \gamma max_{a'} Q_t(s_{now}, a') - Q_t(s_{prev}, a))$
$s_{prev} \leftarrow s_{now}$
return $Q_t$

RETALIATE is controlled by three Q-learning parameters: the "epsilon-greedy" parameter $\varepsilon$, which controls the tradeoff between exploration and exploitation by setting the rate at which the algorithm selects a random action rather than the one that is expected to perform best, the "step-size" parameter $\alpha$, which influences the rate of learning, and the "step-size" parameter $\gamma$, which determines the present value of future rewards. For our empirical evaluations, we found that setting $\varepsilon$ to 0.1, and $\alpha$ to 0.2 work well. RETALIATE diverges from the traditional discounting of rewards by setting $\gamma$ equal to one so that possible future rewards were as important as in selecting the current action as immediate rewards. Initially, we set $\gamma < 1$ to place an emphasis on immediate rewards but found that the rate of adaptation of RETALIATE was slower than when $\gamma$ was set to one.

RETALIATE starts by either initializing all entries in the Q-table with a default value, which was 0.5 in our case study, or by restoring the Q-table from the previous game. The game is then started, and the game state representation $s_{prev}$ is initialized to each domination location having neutral ownership $(N,N,N)$.

The following computations are iterated through until the current game is over. First, the next team action to execute, $a$, is selected using the epsilon-greedy parameter; this means that a random team action is chosen with probability $\varepsilon$ , or the team action with the maximum value in the Q-table for state s is selected with probability 1 - $\varepsilon$. By stochastically selecting actions we ensure that there is a chance of trying new actions, or trying actions whose values are less than the current maximum in

the Q-table. This is important to ensure that RL experiments with a wide range of behaviors before deciding which is optimal.

The selected action $a$ is then executed, and the resulting state, $s_{now}$, is observed. Each bot can either succeed in accomplishing its individual action or fail (e.g., the bot is killed before it could reach its destination). Either way, executing a team action takes only a few seconds because the individual actions are executed in parallel. Updates to the Q-table occur when either the individual actions have completed (whether successfully or unsuccessfully), or domination location ownership changes because of the actions of the opposing team.

Next, the reward $R$ for taking $a$ in $s_{prev}$, is computed as the difference between the utilities in the new state $s_{now}$, and the previous state $s_{prev}$. The reward function, which determines the scale of a reward, is computed as $R = U(s_{now}) - U(s_{prev})$. Specifically, the utility of a state $s$ is defined by the function $U(s) = F(s) - E(s)$, where $F(s)$ is the number of friendly domination locations and $E(s)$ is the number of enemy-controlled domination locations. For example, relative to team *A*, a state in which team *A* owns two domination locations and team *B* owns one domination location has a higher utility than a state in which team *A* owns only one domination location and team *B* owns two.

Finally, the Q-value $Q_t(s_{prev}, a)$ for taking action $a$ in state $s_{prev}$ is used in the standard Q-table update function presented in Equation 1. Having completed the current update, the new state $s_{now}$ is backed up in variable $s_{prev}$ for the next update, and the modified Q-table is returned.

### *3.3* CBRETALITE

One of the limitations of reinforcement learning agents in general and RETALIATE in particular is that the process of converging to an optimal policy may be slow. Worse, when the situation changes in a way that is not reflected directly in the states observed by the agent, a policy that was previously optimal may no longer be a good choice, and the slow process of finding an optimal policy for the new problem must begin. This is a result of the trial-and-error process by which reinforcement learning agents incrementally update their policies based on experience. It is possible to fine-tune the parameters of the Q-learning algorithm to adapt very quickly to changing conditions, but this has its own trade-offs.

Instead, we developed a new system, CBRETALITE, that applies case-based reasoning (CBR) techniques to RETALIATE. The CBRETALITE system stores a library of cases, each of which contains a winning policy and the conditions under which that policy was learned. When the current policy is highly ineffective, the system searches for a case that matches the current situation and begins using the policy from that case. As a result, it is able to quickly change its strategy to counter the different strategies of a dynamic opponent.

### 3.3.1 Case-Based Reasoning

Case-based reasoning is a general problem-solving strategy in which new scenarios are compared with problems that were previously solved, and a successful solution to a previous problem that is similar to the current scenario is adapted to solve the current problem. The knowledge artifacts that store information about previous problem-solving episodes are called *cases*, and typically consist of two components: a representation of the problem that was solved and a representation of the successful solution to the problem. A widely used model of case-based reasoning includes four steps: *Retrieve*, *Reuse*, *Revise*, *Retain*.

In the *Retrieve* step, the system searches for one or more cases in its case library that have a problem that is similar to the current problem. In the *Reuse* step, a solution to the current problem is produced, either as a direct copy of the solution from a retrieved case, or adapted to take into account the differences there may be between the problem in the case and the current case. In the *Revise* step, the system updates its knowledge base as a result of the success or failure of the attempt to solve the current problem with the solution generated in the *Reuse* step. In the *Retain* step, a new case is inserted into the case library. This consists of a problem that was recently solved as well as the solution to that problem. This solution may have been generated from scratch, provided by a tutor, or produced from an existing case. Advanced CBR-based system may also have a feature to manage the case library, such as by removing redundant cases.

### 3.3.2 CBR In CBRETALITE

In CBRETALITE, the solution part of each case is a Q-table, as learned by RETALIATE. The problem representation consists of a set of features describing the current game situation, which are not part of the state representation used within the Q-learning algorithm.

When a DOM game begins there is very limited information available, but CBRETALITE selects a case based on what it does know. The Q-table from that case is used to make decisions during the game, and is updated using the Q-learning algorithm exactly as in RETALIATE. After a certain time window, CBRETALITE determines whether or not it has recently been successful, based on the rate at which each team's score has been changing.

If it has been highly successful, a new case is created and added to the case library. The problem section of this new case consists of the values of the relevant features at this time. The solution section of this new case consists of the Q-table that the agent is currently using, which is likely to have been updated somewhat in the time since it was copied from an existing case.

If CBRETALITE has been very unsuccessful, it will abandon the current Q-table and instead search for a case in the case library that is similar to the current situation and begin using its Q-table. If there is no case sufficiently similar to the current situation, or if CBRETALITE is neither winning nor losing by a significant margin,

then it will neither store nor retrieve a case, but will continue using and updating its current Q-table.

If CBRETALITE has a new case to store and finds that there is already a case in the library with a very similar problem, it will compare how successful the agent was when each of the two cases was created, and will retain only the one with the Q-table that gave the agent the most success.

### 3.3.3 Features And Similarity

The representation of the problem used in the case library consists of several features that, based on observation and trial-and-error, seemed likely to correlate highly with the effectiveness of different strategies. Specifically, CBRETALITE uses the following features:

- *Team Size*: The number of bots on each team.
- *Team Score*: The current score of each team.
- *Bot Distance*: The distance between each bot in the game and each domination location.
- *Ownership*: The fraction of time over a rolling time window in which each domination location was owned by each of the teams.

To compute the level of similarity between one problem and another, CBRETALITE uses a *local similarity metric* for each feature type and a *global similarity metric* that aggregates the values of each local similarity metric. Local similarities are valued between zero and one, and are computed by matching sensory readings from a time window within the current game world with those stored in the case. The value of the aggregate is simply the sum of the local similarity for each feature, divided by the number of features.

The *Team Size* feature type records the number of bots on a team. Teams are assumed to be of equal size, however this assumption could be dropped by using a feature for each team. If $x$ is the size of the team in the current game and $y$ is the team size from a case, $Sim_{Tsize}(x,y)$ is equal to one when $x = y$ and zero otherwise.

The *Team Score* feature type records the score of each team. So, if $x$ is the score of team $A$ in the current game and $y$ is the score of team $B$ from a case, then the similarity is computed by $Sim_{TScore}(x,y) = 1-(|x-y|/SCORE\_LIMIT)$. The constant $SCORE\_LIMIT$ is the score to which games are played. In our case-base, team $A$ is always CBRETALITE and team $B$ is the opponent.

The next feature type, *Bot Distance*, uses the Euclidian distance of each bot to each domination location to compute similarity. That is, each case contains, for each opponent bot $b$ and for each domination location $l$, the absolute value of the Euclidian distance from $b$ to $l$. Specifically, if $x$ is the Euclidian distance of $b$ to $l$ in the current game and $y$ the analogous distance from the case, then $Sim_{Dist}(x,y) = 1-(|x-y|/MAX\_DIST)$. The constant $MAX\_DIST$ is the maximum Euclidian distance any two points can be in a map. With an opposing teams of size 3 and a map with 3 domination locations, there are $3*3 = 9$ of elements of this feature.

The final category of feature, *Ownership*, uses the fraction of time each team $t$ has owned each domination location $l$ during the time window $\delta$ to compute similarity. So, if $x$ is the fraction of time $t$ has controlled $l$ in the current game and $y$ is the analogous fraction from the case, then $Sim_{Own}(x,y) = 1 - |x - y|$. With 2 teams and 3 domination locations, this category has a total of 6 elements.

## 4 Knowledge Representation Requirements

The three agents have some common requirements: an API that allows them to sense information from the game world and send commands to execute in the game world. All agents require a representation of the potential states of the world. The details of the representation vary from agent to agent but basically the state contains information about (1) which team owns each location, (2) where are the bots located, and (3) the score of the game. The agents also require information about the actions that the bots can make. This is basically a state-transition function $S \times A \rightarrow S$ that indicates for each state $s$ and action $a$ what next state will be reached if $a$ is taken in $s$. Aside from these common requirements, some comparisons about each agent's knowledge representation requirements can be made (Table 3 summarizes the three systems).

**Table 3** Knowledge requirements of systems

| Game AI | Description |
|---|---|
| HTNBOTS | Action transition model represented as operators and HTN methods |
| RETALIATE | Action transition model |
| CBRETALITE | Action transition model, features for the cases, similarity metric |

**HTNBOTS has the largest knowledge engineering effort.** HTNBOTS uses the SHOP HTN planning algorithm [10]. SHOP uses a domain-independent algorithm to generate plans that are then executed as outlined in Section 3.1. In order to use SHOP, methods and operators must be provided. As explained in Section 3.1, methods encode how to achieve a compound task whereas operators define valid actions in the domain whereas methods provide knowledge about how to combine the actions to solve problems in the domain (Table 2 shows an example a method and operator). A single operator can describe multiple transitions by using variables; every possible instantiation of variables into constants is one possible transition. Hence, the collection of all operators encodes the state transition function, which as pointed out before, is a common knowledge requirement for all agents. Creating methods requires a deep understanding of the domain to understand the ways in which problems can be solved. The difficulty of creating a list of methods to model completely a domain is a well-known limitation of HTN planning and research has

been conducted aiming at learning this knowledge automatically from a collection of sample plan traces [17, 6, 18].

**RETALIATE has the lowest knowledge engineering effort.** RETALIATE only needs a state transition model (roughly equivalent to the operators in HTN planning) as input. For example, the operator from Table 2 is represented as multiple transitions of the form:

$$(s, sendbot(b, L_D), s') \tag{2}$$

for every bot $b$ and for every pair of states $(s, s')$ such that $s$ is a state where $b$ is in $L_C$ and $s'$ is a state where $b$ is in $L_D$. There is no knowledge about how to combine actions to solve problems because, as explained in Section 3.2, RETALIATE learns this knowledge as policies by using Q-learning.

**CBRETALITE has a low knowledge engineering effort.** In addition to the state transition model of RETALIATE, CBRETALITE needs to identify the features $F$ that will be used to describe the problem section of cases. There are very many possible features of the game world that could be used, and selecting those that are most likely to contain useful information can be more of an art than a science. Additionally, an appropriate local similarity metric for each feature must be identified. Our global similarity metric is a weighted average of the local similarity metrics used for each feature, and setting these weights to an appropriate value is another knowledge engineering challenge.

To offset these burdens, we have also investigated a system to automate much of knowledge engineering process [5]. In that work, we include a large number of features and initially weight them all equally. Through an iterative process, we learn new weights that accurately represent the usefulness of the individual features, and can remove those that have very low weights.

It is possible for a case-based reasoning system to be provided with a case library designed by experts, which would substantially increase the knowledge engineering effort required to use the system. While CBRETALITE will work with cases of any provenance, we have only used cases that it learns through its own experience. Thus, the case library is learned automatically and does not contribute to the knowledge engineering burden.

## 5 Game Performance Comparisons

To test the performance of our AI agents, we pitted them against a variety of hard-coded agents in the DOM game. The performance metric we used is the final score of the agent minus the final score of the opponent. This performance was measured in a variety of maps in a 3-bot versus 3-bot and 4-bot versus 4-bot settings. To determine the effectiveness of learning algorithms, we typically played several games

against the same opponent sequentially, maintaining the knowledge base between them. The opponents were ranked among three classes of teams:

- Easy-difficulty opponents. The team encodes a simplistic strategy that is easy to counter.
- Medium-difficulty opponents. The team encodes a somewhat more difficult strategy to counter.
- Hard-difficulty opponents. The team's strategy is very difficult to counter.

This categorization was obtained through observation of multiple games across different games. The individual behavior of each bot was controlled by the same finite state machine, so this was not a factor for the difference in performance among the teams. We further define opponent strategy as either dynamic or static; dynamic opponents change their strategy over time while static opponents do not. Table 4 presents a summary of the results. For details please refer to the individual papers [3, 16, 1, 7, 9].

Some of the easy-difficulty opponents distributed bots among domination points in a fixed, hard-coded strategy. Others intelligently selected a subset of domination locations to contend for, ordering one bot to defend each and any remaining bots to patrol between them. Medium-difficulty opponents used more complex strategies, such as always sending each bot to the unowned domination point that it is closest to, if any such points exist, or distributing the bots evenly among the domination points that it does not own, without using any to defend the points that it does own. The hard-difficulty opponents used even more complex strategies, or dynamically selected among strategies based on the current situation.

**Table 4** Performance of the three systems

| Game AI | Description |
|---|---|
| HTNBots | Solid performance versus easy- and medium-difficulty opponents. It loses versus hard-difficulty opponents |
| Retaliate | Solid performance versus easy- and medium-difficulty opponents. It wins versus some of the hard-difficulty opponents but loses to others |
| CBRetalite | Improves the performance of Retaliate against easy- and medium-difficulty opponents. |

**HTNBots has a solid performance versus the easy and medium static opponents.** HTNBots was the first system we built to play DOM games. Initially we had a relatively small set of opponents, which over time we found to be easy- and medium-difficult opponents. Against these the initial knowledge base did well. As we added more competent opponents over the years, the performance of HTNBots was poor even against some of the more recent medium-difficulty opponents. This was the result of lack of experience with the DOM game, which meant that the first encodings we created were a bit naïve. Against the most difficult ones, the performance of HTNBots is not as good as the other two agents. We also played

HTNBOTS against RETALIATE and CBRETALITE directly, and it was usually defeated. This revealed some shortcomings in its knowledge base. Recently, the knowledge base of HTNBOTS went through a major overhaul, resulting in significant performance improvement [7, 9]. It now solidly beats all easy- and medium-difficulty opponents from our latest testbed. It is still outperformed by RETALIATE and CBRETALITE when competing versus the most difficult opponents. We believe that further improvements are attainable by modifying the existing HTN methods.

**Over time, RETALIATE achieves a solid performance versus the easy- and medium- static opponents, versus dynamic opponents, and versus some of the most difficult opponents.** RETALIATE was the second game agent we created to play DOM games, and we benefitted from our experiences with HTNBOTS. In particular, we identified a small number of features, such as ownership of the domination locations, that are crucial representatives of the state of the game world. Against easy-difficulty opponents, RETALIATE quickly achieves a good performance (typically very early in the first game). Against medium-difficulty opponents, it will learn a winning policy within the first half of the game. Against some of the difficult opponents, it will still learn a winning policy within the first game. However, against others of the hard-difficulty opponents, it does not seem to be able to converge to a winning policy. We believe that the main factor for this latter behavior is that the current set of features selected is not sufficient to capture all necessary conditions that would allow RETALIATE to counter the strategies of these very difficult opponents. Nevertheless, it is remarkable that RETALIATE is able to learn a winning policy versus most opponents within one game. We also tested RETALIATE versus dynamic opponents (i.e., opponents that change their strategy over time) and it was able to adapt versus these opponents as well.

**CBRETALITE improves the performance over RETALIATE on easy- and medium-difficulty opponents.** CBRETALITE was conceived with the idea of improving on the shortcomings of RETALIATE. Specifically, we expected it to find a winning policy faster than RETALIATE by short-circuiting the RL process in that it would immediately retrieve a "good" policy from the case library and, hence, void the need for RETALIATE to find a winning policy from scratch. For most of the easy- and medium-difficulty opponents CBRETALITE was able to find a winning policy quicker than RETALIATE. This was less so versus the more difficult opponents. In some cases it was able to improve on RETALIATE but the difference between the two was not statistically significant. We believe that the reason for these results is the same as the reason why RETALIATE cannot converge to a winning policy versus some of the hard opponents. Namely, that there are not enough features represented in the state to identify certain situations in the game world where taking one action over another one would be desirable. Still, we found the results promising as CBR helps to address one of the most significant shortcomings of RL by reducing the time the RL algorithm takes to converge to a winning policy.

## 6 Final Remarks

Game AI has received a lot of attention over recent years. There have been a number of works showcasing the use of AI techniques such as reinforcement learning, planning, and case-based reasoning. With very few exceptions, including the Bridge Baron ® and F.E.A.R. ® systems, both of which use AI planning techniques [13, 11], there have been very few fielded applications of these techniques into modern commercial games. Our work showcases some of these difficulties:

- **Difficulty in creating the knowledge bases.** Developing competent players using deliberative reasoning such as HTN planning can require a significant effort to create the knowledge bases. This is consistent with observations about using HTN planning in other domains.
- **Time to generate competent policies.** Learning algorithms such as RL require some time until they converge to competent policies.

At the same time our study points to some promising capabilities. The crucial point is that game researchers have pointed out the need to create competitive AI rather than the most perfect possible one [12, 14]. The goal for common commercial game applications is, after all, not to create an AI that will become unbeatable for a human player but one that is competitive for an average player. With this goal in mind our study points to the following possibilities:

- **Capability to create competent AI.** It is feasible to create good AI with deliberative reasoning techniques such as HTN planning. In fact our experiments show that even the first, somewhat naïve version of the knowledge base could beat all easy-difficulty and some of the medium-difficulty opponents. Further work was sufficient to create a competitive version that could only be beaten by the hard-difficulty opponents.
- **Capability to learn competent AI** It is feasible to learn good AI within reasonable time. Albeit it requires a careful analysis of the features of the state of the game to identify a small subset of these features that is sufficient to guarantee good performance. In addition, CBR can further improve the speed upon which good performance is achieved by skipping several trial-and-error iterations.

For future work, the results of our study points towards an intriguing direction: deliberative AI such as HTN planning could be combined with learning techniques such as combining CBR and RL techniques to attain competent Game AI. Each could be used to address the shortcomings of the other one. HTN planning can start with somewhat competent game AI. This will reduce the knowledge engineering effort compared to creating a knowledge base that is fully competent. At the same time it guarantees a minimum performance level from the outset of the game unlike RL. Using learning techniques one could improve the knowledge base to fill it with newly discovered strategies that the learning algorithm finds while playing. This would address the shortcoming of HTN planning where the initial knowledge base may encode some flawed strategies. There is a challenge with this direction, which

is how to combine the symbolic plan generation process of HTN planning with the stochastic mechanism of RL. In recent work [4] we have begun initial work towards this combination.

**Acknowledgements** This work was sponsored by National Science Foundation (grant #0642882). The views, opinions, and findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the NSF.

## References

1. Auslander, B., Lee-Urban, S., Hogg, C., Muñoz-Avila, H.: 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-08), pp. 59–73. Springer (2008). Trier, Germany
2. Gillespie, K., Karneeb, J., Lee-Urban, S., Muñoz-Avila, H.: Imitating inscrutable enemies: Learning from stochastic policy observation, retrieval and reuse. In: Proceedings of the 18th International Conference on Case-Based Reasoning (ICCBR-10). AI Press (2010)
3. Hoang, H., Lee-Urban, S., Muñoz-Avila, H.: Hierarchical plan representations for encoding strategic game AI. In: Proceedings of the First Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-05). AAAI Press (2005). Marina del Ray, CA
4. Hogg, C., Kuter, U., Muñoz-Avila, H.: Learning methods to generate good plans: Integrating HTN learning and reinforcement learning. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10). AAAI Press (2010)
5. Hogg, C., Lee-Urban, S., Auslander, B., Muñoz-Avila, H.: Discovering feature weights for feature-based indexing of q-tables. In: Proceedings of the Uncertainty and Knowledge Discovery in CBR Workshop at the 9th European Conference on Case-Based Reasoning (ECCBR-08) (2008)
6. Hogg, C., Muñoz-Avila, H., Kuter, U.: HTN-Maker: Learning HTNs with minimal additional knowledge engineering required. In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08). AAAI Press (2008)
7. Muñoz-Avila, H., Aha, D.W., Jaidee, U., Klenk, M., Molineaux, M.: Applying goal directed autonomy to a team shooter game. In: Proceedings of the Twenty-Third Florida Artificial Intelligence Research Society Conference (FLAIRS-10), pp. 465–470. AAAI Press (2010). Daytona Beach, FL
8. Muñoz-Avila, H., Hoang, H.: Coordinating teams of bots with hierarchical task network planning. In: S. Rabin (ed.) AI Game Programming Wisdom 3. Charles River Media, Boston, MA (2006)
9. Muñoz-Avila, H., Jaidee, U., Aha, D.W., Carter, E.: Goal directed autonomy with case-based reasoning. In: Proceedings of the Eighteenth International Conference on Case-Based Reasoning (ICCBR-2010). Springer (2010)
10. Nau, D.S., Cao, Y., Lotem, A., Muñoz-Avila, H.: SHOP: Simple hierarchical ordered planner. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), pp. 968–973. AAAI Press (1999). Stockholm
11. Orkin, J.: Three states and a plan: The A.I. of F.E.A.R. In: Proceedings of the Game Developer's Conference (GDC) (2006)
12. Scott, B.: The illusion of intelligence. In: AI Game Programming Wisdom. Charles River Media (2002)
13. Smith, S.J.J., Nau, D., Throop, T.A.: A planning approach to declarer play in contract bridge. Computational Intelligence **12**(1), 106–130 (1996)

14. Spronck, P., Sprinkhuizen-Kuyper, I., Postma, E.: Difficulty scaling of game AI. In: A.E. Rhalibi, D.V. Welden (eds.) Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAME-ON-04) (2004). EUROSIS, Belgium
15. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998)
16. Vasta, M., Lee-Urban, S., Muñoz-Avila, H.: RETALIATE: Learning winning policies in first-person shooter games. In: Proceedings of the Seventeenth Innovative Applications of Artificial Intelligence Conference (IAAI-07). AAAI Press (2007)
17. Yang, Q., Wu, K., Jiang, Y.: Learning action models from plan traces using weighted MAX-SAT. Artificial Intelligence Journal (AIJ) **171**(2-3) (2007)
18. Zhuo, H.H., Hu, D.H., Hogg, C., Yang, Q., Muñoz-Avila, H.: Learning HTN method preconditions and action models from partial observations. In: Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09). AAAI Press (2009)

# Index