Recognizing the Enemy: Combining Reinforcement Learning with Strategy Selection using Case-Based Reasoning

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

Dept. of Computer Science & Engineering
Lehigh University
Bethlehem, PA, USA

Abstract. This paper presents CBRetaliate, an agent that combines Case-Based Reasoning (CBR) and Reinforcement Learning (RL) algorithms. Unlike most previous work where RL is used to improve accuracy in the action selection process, CBRetaliate uses CBR to allow RL to respond more quickly to changing conditions. CBRetaliate combines two key features: it uses a time window to compute similarity and stores and reuses complete Q-tables for continuous problem solving. We demonstrate CBRetaliate on a team-based first-person shooter game, where our combined CBR+RL approach adapts quicker to changing tactics by an opponent than standalone RL.

1 Introduction

Reinforcement Learning (RL) has been successfully applied to a variety of domains including game theoretic decision processes [1] and RoboCup soccer [2]. It has also been applied successfully for a number of computer gaming applications including real-time strategy games [3], backgammon [4], and more recently for first-person shooter (FPS) games [5].

Despite these successes, it may take a while before an agent using RL adapts to changes in the environment. This is the result of the exploration process, in which the agent must try new actions with unknown utility to develop a policy maximizing its expected future rewards. This can be problematic in some applications. For example, we observed this when applying RL techniques to team-based first-person shooters (TFPS). TFPS is a very popular game genre where teams of two or more players compete to achieve some winning conditions. In TFPS games, individual players must have good reflexes to ensure short-term survival by shooting the enemy and avoiding enemy fire while working together to achieve the winning conditions of the game. In recent work we constructed an agent, Retaliate, which uses an online RL algorithm for developing winning policies in TFPS games [5]. Specifically, Retaliate uses the Q-learning variant of RL, in which a policy is encoded in a table of expected rewards for each state-action pair, called a Q-table. Retaliate demonstrated that it was capable of developing a winning policy very quickly within the first game against an opponent that

used a fixed strategy. We also observed that it took Retaliate a number of iterations before it adapted when the opponent changed its strategy. Thus, we began considering techniques that would allow us to speed up the adaptation process in such situations where the strategy employed by an opponent changes.

In this paper we present CBRetaliate, an agent that uses Case-Based Reasoning (CBR) techniques to enhance the Retaliate RL agent. Unlike most previous work where RL is used to improve accuracy in the case selection process, CBRetaliate uses CBR to jump quickly to previously stored policies rather than slowly adapting to changing conditions. Cases in CBRetaliate contain features indicating sensory readings from the game world when the case was created. They also store the complete Q-table that is maintained by CBRetaliate when the case was created. CBRetaliate stores a case when it has been accumulating points at a faster rate than its opponent during a time window. When it is accumulating points more slowly than its opponent, it attempts to retrieve a similar case. CBRetaliate uses an aggregated similarity metric that combines local similarity metrics for each feature. This similarity metric is computed by matching sensory readings from the current gaming world and those of the case over the time window. When a case is retrieved, its associated Q-table is adapted by Retaliate by using standard RL punishment/reward action selection.

Our working hypothesis is as follows. The use of CBR will allow CBRetaliate to recognize strategies similar to ones it has faced previously but different from the one it has most recently fought, and thus to outperform Retaliate when such a strategy change occurs. We tested our hypothesis with an ablation study comparing the performance of Retaliate and CBRetaliate in games against a number of opponents each using a different strategy. Each of these tests consisted of a tournament of several consecutive games with the Q-table saved between games. Within a tournament, CBRetaliate was able to more soundly beat an opponent similar to one it had previously faced by loading a case learned from the previous opponent. The nature of its opponent was not defined for CBRetaliate, but needed to be inferred from sensory readings describing the behavior it observed over time.

The paper continues as follows: the next section describes the TFPS game and the Retaliate algorithm. Next, in Section 3, we describe CBRetaliate by discussing how it uses the phases of the CBR problem-solving cycle. The next section describes the empirical evaluation. Section 5 presents related work. We conclude this paper with some final remarks.

2 Background

The CBRetaliate agent is an extension of an existing Reinforcement Learning agent, Retaliate, to use techniques from Case-Based Reasoning. As a testbed for this agent, we use a configuration of a first-person shooter game in which individual computer-controlled players (bots) act independently but follow a team-level strategy to achieve their objectives.

2.1 Domination Game Domain

Unreal Tournament (UT) is a first-person shooter game in which the usual objective is to shoot and kill opposing players. Players track their health and their weapon's ammunition, as well as attempt to pick up various items strewn about the map while amassing kills and preserving their own life. Opponents may be other human players via online multiplayer action or computer-controlled bots. An interesting feature of UT is the ability to play several different game variants. One of these variants is a domination game, a feature offered by many team-based multiplayer game.

In a domination game, the player's objective is not to earn kills, although this is usually necessary. Rather, the goal is to accumulate points for a player's team by controlling certain locations in the game world known as domination locations. A domination point is controlled by the team of the player who was last in the location, and lost when a player from the opposing team reaches it. Each domination point produces points over time for the team that controls it, and the game ends when one team's score reaches some threshold.

Domination games are ideal test domains for cooperative artificial intelligence agents because they require both tactics to succeed in individual firefights and strategy to decide how and where individual bots should be deployed. We have chosen to focus exclusively on strategy, using an abstract model described in Section 2.4.

2.2 HTNbots

One of the first successful agents developed for controlling teams of bots in UT domination games was HTNbots [6]. HTNbots uses Hierarchical Task Network (HTN) planning to generate plans during the game. The preconditions of HTN methods used by HTNbots map to state information about the game world, and the operators correspond to commands telling each individual bot where it should attack or patrol. We now use HTNbots as a known difficult opponent against which Retaliate and CBRetaliate can be compared.

2.3 Retaliate

Retaliate is an online RL algorithm for developing winning policies in team-based first-person shooter games. Retaliate has three crucial characteristics: (1) individual bot behavior is fixed although not known in advance, therefore individual bots work as plugins, (2) Retaliate models the problem of learning team tactics through a simple state formulation, (3) discount rates commonly used in Q-learning are not used. As a result of these characteristics, the application of the Q-learning algorithm results in the rapid exploration towards a winning policy against an opponent team. In our empirical evaluation we demonstrate that Retaliate adapts well when the environment changes.

Retaliate is controlled by two parameters: ε, which is known as the "epsilon-greedy" parameter and controls the trade-off between exploration and exploitation by setting the rate at which the algorithm selects a random action rather

Algorithm 1 RetaliateTick(Q_t)

Input: Q-Table Q_t
Output: updated Q-table
ε is .10, and $State_{prev}$ is maintained internally
if $\mathsf{rand}(0,1) > \varepsilon$ **then** {epsilon greedy selection}
  $Act \leftarrow$ applicable action with max value in Q-table
else
  $Act \leftarrow$ random applicable action from Q-table
$State_{now} \leftarrow \mathsf{Execute}(Act)$
$Reward \leftarrow \mathsf{Utility}(State_{now}) - \mathsf{Utility}(State_{prev})$
$Q_t \leftarrow$ update Q-table
$State_{prev} \leftarrow State_{now}$
return Q_t

than the one that is expected to perform best, and γ, which is referred to as the "step-size" parameter and influences the rate of learning. For our case study, we found that setting ε to 0.1 and γ to 0.2 worked well.

The following computations are iterated through until the game is over. First, the next team action to execute, *Act*, is selected using the epsilon-greedy parameter. The selected action *Act* is then executed.

On the next domination ownership update from the server, which occurs rougly every four seconds, the current state $State_{now}$ is observed and the Q values for the previous state $State_{prev}$ and previously selected actions are updated based on whether or not $State_{now}$ is more favorable than $State_{prev}$. New actions are selected from the new current state, and the process continues.

The reward for the new state $State_{now}$ is computed as the difference between the utilities in the new state, and the previous state $State_{prev}$. 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 that are controlled by the enemy. This has the effect that, relative to team A, a state in which team A owns two domination locations and team B owns one has a higher utility than a state in which team A owns only one domination location and team B owns two. The reward function, which determines the scale of the reward, is computed as $R = U(State_{now}) - U(State_{prev})$.

The calculated 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 set of actions a were ordered. This calculation is performed according to the following formula, which is standard for computing Q-table entries in temporal difference learning [7]:

$Q(s, a) \leftarrow Q(s, a) + \alpha(R + \gamma \times max_{a'} Q(s', a') - Q(s, a))$

In this computation, the entry in the Q-table for the action a that was just taken in state s,$Q(s, a)$, is updated. The function $max_{a'}$ 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 γ, which is called the discount factor parameter, adjusts the relative influences of current and future rewards in the decision making process.

2.4 Game Model

The Q-learning algorithm on which Retaliate is based stores the expected future reward of each potential action in each state. There are many potential features that could be used to define the state of the game and numerous actions a bot may take at various levels of granularity. In Retaliate, we chose to use a very simple, abstract model of the game world. Specifically, each state is defined by the current ownership of each domination point. For a game containing three domination points and two teams, as in our experiments, each state is a 3-tuple where each value is either "Friendly", "Enemy" or "Unowned" (the default before any bot has entered the location). Thus, such a game has 27 possible states.

Because we are focusing on grand team strategy rather than tactics, our action model is similarly simple. Each action consists of the assignments of each bot on the team to one of the domination points. Thus, a game with three domination points and teams of three bots will similarly have 27 possible actions.

This model of the world is quite simple, but surprisingly effective. Enough information is provided to allow the representation of a robust strategy and the Q-table is small enough that the algorithm is able to converge to a reasonably complete table within the space of only a few games.

3 Algorithm

When the situation changes so dramatically that the policy encoded by Retaliate is no longer valid, such as by changing the opponent, the Q-learning algorithm must slowly explore the policy space again, trying actions and updating the rewards until it finds a new good policy. We developed CBRetaliate to solve this problem by storing winning policies and retrieving them later based on other types of features from the game state. In this section we present the contents of cases, how similarity is computed, and finally the psuedocode for CBRetaliate.

3.1 Case Features and Similarity Functions

As stated previously, CBRetaliate uses an aggregated similarity metric that combines the local similarity metrics for each case feature. 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. We found CBRetaliate to be effective with this naive aggregate function and feature weights, but expect that much better performance would be possible if these parameters were carefully tuned.

Each case contains a Q-table along with a set of features that are summarized in Table 1. The first two categories of features, *Team Size* and *Team Score* are notable because they do not involve the navigation task. Whereas our RL problem model is limited to domination location ownership in order to reduce the

Category	Description	Local Sim. Function
Team Size	The number of bots on a team.	Sim_{TSize}
Team Score	The score of each team	Sim_{TScore}
Bot/Dom Dist.	Distance of each bot to each dom. loc.	Sim_{Dist}
Dom Ownership	Which team owns each of the dom. locs	Sim_{Own}

Table 1. Description of feature categories and their local similarity function name

state space, the CBR component does not share this restriction. Consequently, the name of each team as well as the map name could have been used as features, however, we wished to demonstrate the ability of CBRetaliate to recognize strategies and situations based on behavior and observations.

The *Team Size* category is currently a single feature that records the number of bots on a team. Teams are assumed to be of equal size, however this assumption could be easily dropped by adding 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* category consists of two features, namely 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 and is 100 in our experiments. In our case-base, team A is always CBRetaliate and team B is the opponent.

The next category of features, *Bot/Dom Dist.*, 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 an Unreal Tournament map. With an opposing teams of size 3 and a map with 3 domination locations, this category has a total of $3 * 3 = 9$ features.

The final category of features, *Dom Ownership*, uses the fraction of time each team t has owned each domination location l during the time window δ (elaborated upon in the next subsection) 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 $Sim_{Own}(x, y) = 1 - |x - y|$. Intuitively, with 2 teams and 3 domination locations, this category has a total of 6 features.

3.2 The CBRetaliate Algorithm

Algorithm 2 shows at a high-level how CBRetaliate operates during a single game. However, before explaining the algorithm, we must first define four constants that control its behavior.

The first constant, U^l, defines the minimum number of game cycles that must occur, since the last case was retrieved or retained, before the load of a case is considered. During retrieval the best case is returned and is used only if its similarity is above the second constant, $THRESH$. The third constant, U^s, has the same meaning as U^l except controls when saving can occur. For our empirical evaluation we used $U^l = 22$, $U^s = 30$, and 0.75 for $THRESH$.

The fourth and final constant, δ, is used in two important ways. On the one hand, δ is used to determine whether or not CBRetaliate is accumulating points faster than its opponent by computing the current difference in score at game cycle t and subtracting from that the score difference at cycle $t-\delta$. On the other hand, δ is also used to compute the so-called "sliding average" of domination location ownership. This average tracks, for each domination location l, the fraction of time that each team has owned l within the window defined between the current game cycle t and $t - \delta$ (this value is used in Sim_{Own}). For our empirical evaluation, we set δ to 15.

Algorithm 2 CBRetaliate(CB, Q_t)

```
1: Input: case-base CB, Q-table Q_t
2: Output: The updated CB, and the Q-table last loaded Q_t
3: num_updates ← 0
4: while game is not over do
5:    num_updates++
6:    Q_t ← RetaliateTick(Q_t) {Revise}
7:    S_now ← GetCurrentState
8:    if num_updates >= δ then {wait for window}
9:       if (ScoreDiff_now − ScoreDiff_now−δ) > 0 then
10:         if num_updates >= U^s then {enough Q-table updates}
11:            CB ← SaveCase(Q_t, CB, S_now) {Retain}
12:            num_updates ← 0
13:      else
14:         if num_updates >= U^l then {enough Q-table updates}
15:            SimCase ← OnePassRetreive(S_now) {find most sim case}
16:            if similarity(S_now,SimCase) > THRESH then {similar enough}
17:               Q_t ← getQTable( SimCase ) {Reuse}
18:               num_updates ← 0
19: return (CB, Q_t)
```

Algorithm 2 works as follows. When started for the first time, the case-base CB is empty, and every entry in the Q-table is initialized to the same default value. During a game, the number of game cycles that have passed since the last case load or save is tracked with the variable $num\_updates$. In line 6, algorithm 1 is used to update the Q-table on every game cycle, as explained in Section 2.3. Line 8 ensures that there have been at least δ game cycles since the last case was loaded or saved before allowing the algorithm to proceed. As a consequence of waiting at least δ game cycles, the Retaliate algorithm is able to perform at

least a few Q-table updates before an alternate table is considered. This helps avoid reloading tables when losing, and also gives Retaliate a chance to learn a better strategy.

If enough cycles have occurred, line 9 computes whether or not CBRetaliate has increased its winning margin in the last δ updates. If the winning margin has increased, and there have been a sufficient number of game cycles (U^s), the current Q-table is added to the case-base, along with all features describing the current game state (S_{now}), and $num\_updates$ is reset. A save when the winning margin has increase is sensible because the Q-table in use is clearly working well against the opponent. Otherwise, if the winning margin has decreased and there have been a sufficient number of game cycles (U^l), the case in the case base most similar to the current game features is retrieved. If the similarity of the retrieved case is above $THRESH$, the Q-table from that case is used to replace the Q-table currently-in-use and $num\_updates$ is reset.

4 Evaluation

To evaluate the effectiveness of combining Case-Based Reasoning with Reinforcement Learning in this way, we have performed several experiments using the technique to control teams of bots in domination games. It should be noted that we found a bug that gives the learning teams an advantage over non-learning teams. However, this glitch does not effect our claims of using CBR with RL, because both CBRetaliate and Retaliate are learning teams.

4.1 Evaluation Against CompositeBot

In order to easily test our hypothesis about an opponent that changes strategies, we developed a simple configurable agent called CompositeBot. CompositeBot does not use any information about the game state, but simply provides static assignments of each team member to a domination point. Rather than changing strategies within a single game, we ran a series of seven games consecutively, changing the configuration of CompositeBot (its static assignments) between each game. The map on which these games were played contains three domination points that we will call "A", "J", and "R".

In the first three games, we configured CompositeBot to use a strategy of stationing two bots at one of the domination points and one at another, changing the points selected between games. The next three games are repeats of the first three. In the last game, the opponent sends one bot to each domination point. The specific strategies used in each game are shown in Table 2.

We ran 15 trials each of both Retaliate and CBRetaliate against this series of opponents. Each trial begins with an empty Q-table and (for CBRetaliate) an empty case base. Both the Q-table and case base are updated and enhanced throughout the course of the 7 games.

The results of this experiment are summarized in Table 3. Each game ends when one of the team reaches 100 points. All results are an average over the 15

Game	1	2	3	4	5	6	7
Strategy	AAJ	RRA	JJA	AAJ	RRA	JJA	AJR

Table 2. CompositeBot configurations

		Difference At 25%	**Different At 100%**
Game 1	Retaliate	7.72	53.57
	CBRetaliate	8.10	52.93
Game 2	Retaliate	**9.7**	48.35
	CBRetaliate	**6.01**	46.49
Game 3	Retaliate	**6.96**	**47.75**
	CBRetaliate	**11.18**	**68.49**
Game 4	Retaliate	**6.02**	**57.8**
	CBRetaliate	**10.05**	**65.84**
Game 5	Retaliate	8.37	37.54
	CBRetaliate	7.5	49.11
Game 6	Retaliate	6.53	58.66
	CBRetaliate	7.92	62.98
Game 7	Retaliate	**3.40**	53.01
	CBRetaliate	**10.1**	58.35

Table 3. CompositeBot results

trials. The values in this table are the difference in score between the algorithm being tested and its opponent when the game is 25% finished and when it is complete. Differences that are statistically significant with a 90% confidence level are bolded.

One of the motivations for this work was an expectation that CBRetaliate would have much better early performance than Retaliate when facing an opponent from which it had already stored cases, because it would be able to immediately jump to a Q-table that had been effective against the opponent in the past. Thus, we would expect CBRetaliate to perform significantly better than Retaliate in the first 25% of games 4, 5, and 6. Although this is the case in games 3, 4, and 7, it is not true of 5 or 6. Furthermore, Retaliate has an early advantage in the second game. There are two reasons why we have not consistently seen this expectation met. First, the features used for case retrieval require trend information about the game. Thus, it is difficult to reliably select a good case until enough of the game has been played to recognize the opponent's strategy. The other contributing factor is that the locations of the domination points are not known at the beginning of the game, and strategies cannot be used until the bots have discovered them by exploring the map. We do not explicitly count the exploration phase as a team action, but rather treat it as an initialization phase because all teams use the same search algorithm for the same length of time. Work is underway to remove the need for finding locations. All domination

points are found, on average, when 13% of the game is finished, but in rare cases there have been games that end before all have been found.

In game 1, Retaliate and CBRetaliate perform nearly identically by the end of the game. This is expected, because when CBRetaliate has no cases stored it works exactly like Retaliate (except that it stores new cases). Figure 1 shows the comparative performance of Retaliate and CBRetaliate in the first game. This and all future graphs show the difference between the scores of each algorithm and its opponent over time, which is scaled to the percentage of game finished to facilitate averaging over several trials.

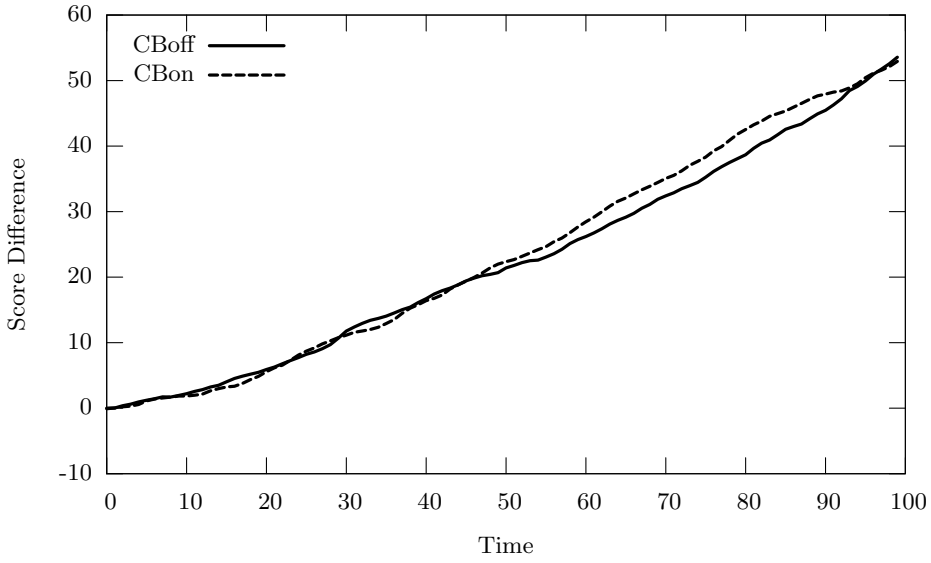

Fig. 1. Averaged score differential in game 1

Retaliate gains a small advantage in the second game, but is beaten soundly in the third. At the start of game 3, Retaliate will have a mature Q-table built to counter a strategy that heavily defends domination point "R", lightly defends point "A", and ignores "J". Such a Q-table will be poorly suited to fighting an opponent who heavily defends "J", lightly defends "A", and ignores "R". Retaliate is able to win in spite of its poor initial strategy by adapting and favoring those decisions that have positive outcomes. CBRetaliate, however, loads a Q-table from the end of the first game. The strategies of the opponents in the first and third games are not identical, but they are similar enough that a strategy effective against one will be somewhat effective against the other.

CBRetaliate wins by a smaller but still significant margin in game 4, where it faces an opponent identical to the one from game 1. The score differentials from

this game are shown in Figure 2. In this case Retaliate should have a reasonable strategy from the previous game, but CBRetaliate is able to load an excellent strategy from the first game. On average, CBRetaliate wins by a similar margin in games 5 and 6, but these results are not statistically significant due to higher variance. CBRetaliate also does well against the balanced strategy of game 7, even though it has not previously faced that strategy. This is because it returns to a less mature Q-table from the early parts of a previous game that is more suited to combating a balanced strategy than the specialized Q-table that Retaliate starts with.

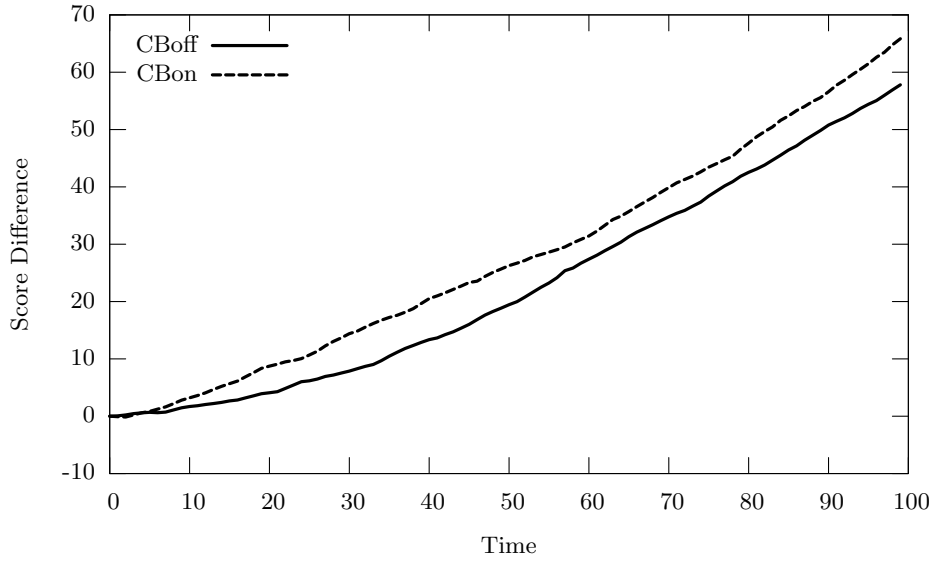

Fig. 2. Averaged score differential in game 4

4.2 Evaluation Against HTNbots

We also performed a second experiment in which CBRetaliate and Retaliate were matched against HTNbots. For this experiment, we used a sequence of 10 games. We did not alter HTNbots between games, but expected that its natural ability to choose different strategies would allow it to perform better against Retaliate than against CBRetaliate.

Surprisingly, this was not the case. The only stastically significant difference between the performance of Retaliate and CBRetaliate against HTNbots was in game 8, where Retaliate won by a higher margin. Across all 10 games, Retaliate beat HTNbots by an average of 22.73 points while CBRetaliate's margin of victory was 23.86 points, a nearly indistinguishable difference.

The reason for these results is a design flaw with the knowledge base encoded in HTNbots that was only revealed through these experiments. HTNbots has one strategy used when not all domination points have been found and one strategy for each number of domination points it controls when the locations of all are known. Ownership of domination points can change quite rapidly during a competitive game, causing HTNbots to quickly oscillate between strategies as it loses and retakes domination points. CBRetaliate is designed to respond to significant, long-lasting changes in strategy. Thus, it retrieves cases based on observed behavior over a time interval. If the opponent is frequently changing strategies such that throughout most of the game it is using its control-one strategy 60% of the time and its control-two strategy 40% of the time, then this combination is effectively a single static strategy, and CBRetaliate will have no significant advantage over Retaliate.

5 Related Work

There are a number of works combining Case-Based Reasoning and Reinforcement Learning. In his ICCBR-05 invited talk, Derek Bridge pointed out that one of the possible uses of such a combination is for continuous problem solving tasks [8]. Winning domination maps in an FPS game is precisely an example of such a task. Our approach fits in Bridge's 11-step CBR problem solving cycle; policies are retrieved based on continuous sensory input. These policies are reused and refined with RL updates while affecting the environment. These policies are then retained, together with current sensory measurements, as new cases.

The CAT system [9] stores and reuses cases having sequences of scripting commands in a real-time strategy game. For retrieval purposes, these cases are annotated with the conditions observed when the case was stored. These conditions include the current research level in the game (which influences which buildings and units can be constructed) and several conditions that compute the difference between CAT's controlled player and the opponent's controlled player (e.g., the number of enemy buildings destroyed minus the number of friendly buildings destroyed by the enemy). When a case it retrieved, its sequence of scripting commands is executed. There are three key differences between CBRetaliate and CAT. First, retrieval in CBRetaliate is performed based on sensory readings from a δ-time window $[t - \delta, t]$ rather than readings at a time t as in CAT. Second, CBRetaliate stores a Q-table, which contains the strategy to be followed and alternative strategies, rather than a sequence of scripted actions. A policy can be seen as representing multiple sequences of scripted actions. Third, in CAT, the case's scripted actions are not adapted. In CBRetaliate, the retrieved Q-table is adapted with the standard reward and punishment operations of RL.

In [10], a CBR system capable of playing real-time strategy games is presented. The system learns cases by observing users' actions. It reuses cases by combining them into strategies that consists of the combination of individual cases. In contrast, CBRetaliate stores Q-tables as cases, which contain the winning strategy together with alternative strategies.

The CARL architecture combines CBR and RL to create agents capable of playing real-time strategy games [11]. CARL is a multi-level architecture similar in spirit to hierarchical task network representations [12] where the higher levels of the hierarchy represents strategies and the low level concrete actions. At the highest level a hand-coded planner is used. At the intermediate level, CBR and RL are used to select the specific tactic (e.g., to attack, to defend), and at the concrete level a plan executor module controls the actions being executed. As a comparison, CBRetaliate can be seen as a two-level architecture. At the top level CBR and RL are used to learn and reuse the strategy to follow. At the bottom level, bots follow these strategies using hard-coded programs. This difference is not arbitrary but almost certainly a design decision that reflects the difference between the two game genres that each system is targeting. In first-person shooters, targeted by CBRetaliate, fast reflexes are needed from individual bots, as players need to respond in fractions of a second to attacks from an opponent or make quick decisions to grab a nearby weapon or follow an opponent. Therefore, in CBRetaliate individual bot behavior is hard-coded. In real-time strategy games, players have more time (seconds at least) to decide if they are going to attack or defend. Like in CBRetaliate, cases in CARL stored what amounts to a Q-table, annotated with the applicability conditions. But unlike CBRtaliate but as with CAT, case retrieval in CARL is based on mapping of current readings at time t rather than in a time window $[t - \delta, t]$ as in CBRetaliate.

CBRetaliate is closely related to *Continuous Case-Based Reasoning*, which was implemented in the SINS system for robot navigation tasks [13]. Continuous CBR advocates that in domains involving real time execution, a time window or *time-history*, as originally called, should be considered during retrieval. All features in SINS are numerical, reflecting the navigation domain targeted. Thus, the difference in trajectories is reflected in the computation of similarity. CBRetaliate also uses features that reflect geometrical relations in the map (e.g., distance between a bot and a domination location). However, CBRetaliate also uses features that are not geometrical relations (e.g., the current score in the game). As a result, we needed to use an aggregate similarity metric to combine these distinctive local similarity metrics. Another difference is that SINS did not use RL for adapting the navigation path. This is possibly due to the fact that a direct application of RL would have resulted in a large search space. More recent work on robotics have found ways to work around that problem (e.g., [14]).

Researchers have proposed to use domain knowledge encoded as HTNs or similar representations in the context of RL and more generally MDPs [15, 16]. One of the results of combining HTN-like knowledge and RL/MDPs is a significant reduction in the search space compared to standalone RL/MDPs. The reason for this is that knowledge encoded in the HTN eliminates unneccesary parts of the search space, parts which *pure* RL/MDPs approaches would otherwise need to explore. In CBRetaliate we do not provide such knowledge in advance, so it is conceivable that CBRetaliate could also benefit from search reduction, albeit with the tradeoff of extra effort required to encode the domain knowledge.

6 Conclusions

It is possible to enhance the states as defined currently in Retaliate by adding the 18 features currently used in CBRetaliate to compute case similarity to the 3 features already used by Retaliate. This would require discretizing the real-valued attributes and vastly increasing the number of states in the Q-table. Rather than using such an expanded table, which would pose technical challenges and require far more time to become mature, CBRetaliate can be seen as partitioning the space of possibilities into regions, each with a suitable Q-table associated with it, and using CBR to "jump" to the appropriate region of the space by selecting a suitable 27-cell table for that region. In our experiments, this capability of CBRetaliate to jump between regions demonstrated speed-up in the elicitation of a winning policy when the opponent was changed.

Another point to be made is that we applied in our experiments a naive approach when computing the local similarities. For each feature, local similarity is basically defined as a linear interpolation between the lowest and the highest possible distance between pairs of values for that feature. Furthermore, no weights were used when aggregating these local similarities to compute a global similarity metric. Significant gains in accuracy of the retrieval process can be made if we use feature weighting, which could be computed by using statistical sampling. The same can be said with the retrieval threshold. It was set to 75% in our experiments and this value was selected arbitrarily. Retrieval accuracy could be improved by tuning the threshold. The reason for not doing any of these possible improvements is that we wanted to test our working hypothesis without tweaking these parameters, so that we could confidently attribute the results to the CBR approach rather than to some tweaking of these parameters.

In this paper we presented CBRetaliate, a CBR + RL system that is intended to enhance RL capabilities for situations in which the environment suddenly changes. CBRetaliate uses time windows during case retrieval and retention. It stores and retrieves Q-tables to allow the RL algorithm to rapidly react to changes in the environment. We demonstrated our approach in a TFPS game, which is characterized by the speed in the decision making by individual bots and in the overall strategy. Our results demonstrate that CBR can effectively speed-up the RL adaptation process in dynamic environments.

As future work, we want to study case-base maintenance issues in the context of CBRetaliate. In the experiments reported in this paper, we reset the case base at the beginning of each tournament. As a result, the retrieval times were very low and did not have any effect on the overall performance of the agent. Clearly this will change in situations when the case base becomes permanent, and a mechanism to refine the case base will be necessary. This poses some interesting research questions: (1) Because cases contain Q-tables, how can we tell if a case is covered by another case? (2) As the Q-table of a retrieved case is updated with RL, can we identify situations where the updated Q-table should replace one of the retrieved cases instead of being stored as a new case as currently done by CBRetaliate? We intend to address these and other questions in the future.

Acknowledgments

This research was in part supported by the National Science Foundation (NSF 0642882).

References

1. Bowling, M.H., Veloso, M.M.: Multiagent learning using a variable learning rate. Artificial Intelligence **136**(2) (2002)
2. Salustowicz, R.P., Wiering, M.A., Schmidhuber, J.: Learning team strategies: Soccer case studies. Mach. Learn. **33**(2-3) (1998)
3. Ponsen, M., Spronck, P.: Improving adaptive game AI with evolutionary learning. In: Proceedings of Computer Games: Artificial Intelligence, Design and Education (CGAIDE-04). (2004)
4. Tesauro, G.: Temporal difference learning and TD-Gammon. Communications of the ACM **38**(3) (1995)
5. Smith, 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)
6. Hoang, H., Lee-Urban, S., Muñoz-Avila, H.: Hierarchical plan representations for encoding strategic game AI. In: Proceedings of the first Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05), AAAI Press (2005)
7. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998)
8. Bridge, D.: The virtue of reward: Performance, reinforcement and discovery in case-based reasoning. Invited Talk at the 6th International Conference on Case-Based Reasoning (ICCBR-05) (2005)
9. Aha, D.W., Molineaux, M., Ponsen, M.J.V.: Learning to win: Case-based plan selection in a real-time strategy game. In: Proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR-05). (2005)
10. Ortañón, S., Mishra, K., Sugandh, N., Ram, A.: Case-based planning and execution for real-time strategy games. In: Proceedings of the 7th International Conference on Case-Based Reasoning Research and Development (ICCBR-07). (2007)
11. Sharma, M., Holmes, M., Santamaría, J.C., Irani, A., Jr., C.L.I., Ram, A.: Transfer learning in real-time strategy games using hybrid CBR/RL. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07). (2007)
12. Erol, K., Hendler, J., Nau, D.S.: HTN planning: complexity and expressivity. In: Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94). (1994)
13. Ram, A., Santamaria, J.C.: Continuous case-based reasoning. Artificial Intelligence **90**(1-2) (1997)
14. Ros, R., Veloso, M.M., de Màntares, R.L., Sierra, C., Arcos, J.L.: Retrieving and reusing game plays for robot soccer. In: Proceedings of the 8th European Conference on Advances in Case-Based Reasoning (ECCBR-06). (2006)
15. Kuter, U., Nau, D.: Using domain-configurable search control in probabilistic planners. In: Proceedings of the The Twentieth National Conference on Artificial Intelligence (AAAI-05). (2005)
16. Ulam, P., Goel, A., Jones, J., Murdock, J.W.: Using model-based reflection to guide reinforcement learning. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05) Workshop on Reasoning, Representation and Learning in Computer Games. (2005)