

Imitating Inscrutable Enemies: Learning from Stochastic Policy Observation, Retrieval and Reuse

Kellen Gillespie, Justin Karneeb, Stephen Lee-Urban, Héctor Muñoz-Avila

Department of Computer Science and Engineering, Lehigh University,
19 Memorial Drive West, Bethlehem, PA 18015 USA
{kkg210, jtk210, sml3, hem4}@lehigh.edu

Abstract. In this paper we study the topic of CBR systems learning from observations in which those observations can be represented as stochastic policies. We describe a general framework which encompasses three steps: (1) it observes agents performing actions, elicits stochastic policies representing the agents' strategies and retains these policies as cases. (2) The agent analyzes the environment and retrieves a suitable stochastic policy. (3) The agent then executes the retrieved stochastic policy, which results in the agent mimicking the previously observed agent. We implement our framework in a system called JuKeCB that observes and mimics players playing games. We present the results of three sets of experiments designed to evaluate our framework. The first experiment demonstrates that JuKeCB performs well when trained against a variety of fixed strategy opponents. The second experiment demonstrates that JuKeCB can also, after training, win against an opponent with a dynamic strategy. The final experiment demonstrates that JuKeCB can win against "new" opponents (i.e. opponents against which JuKeCB is untrained).

Keywords: learning from observation, case capture and reuse, policy.

1 Introduction

Children learn by observing and then mimicking the actions taken by adults or other children. From this psychological motivation, learning from observation has become an important and recurrent topic in AI [1]. In a nutshell, an agent observes the actions performed by another agent while solving a problem and reuses those actions while solving problems. Case-based reasoning systems can be seen as learning from observations; cases retain previously observed episodes and are reused when solving new problems [2].

In the context of agents interacting in adversarial environments, like games, a number of representations have been studied for CBR systems to encode an agent's observations. Table 1 summarizes these representations. We identified four representations for the case (the representations refer to the "solution" part of the case; this is the part of the case that is reused to solve new problems). These representations are listed in order of generality, from the less to the more general (the more general can represent the less general, but the opposite is not necessarily true):

(1) a single action/command, (2) a script (i.e., a sequence of commands), (3) a plan (i.e., a sequence of actions) and (4) a policy (i.e., a mapping from state to actions, indicating for each state s the action that the agent must take). We distinguish between scripts and plans. The commands in the scripts have no explicit semantics for the CBR system. In contrast, the actions in plans have explicit semantics as indicated by the action’s preconditions and effects [3].

Table 1. Representations used for adversarial CBR

Representation	Description	Sample System/Paradigm
action	A single action/command	SIBL
scripts	Sequences of commands	CAT
plans	Sequences of actions	Darmok
policy	A map: States \rightarrow Actions	CBRetaliate

Epstein and Shih [4] present an example of a CBR system that uses sequential instance-based learning (SIBL) to decide an agent’s next move in playing bridge; these actions are captured from previous instances based on the sequence of states visited so far. Case-based Tactician (CAT) [5] uses scripts to indicate the sequence of commands to execute. CAT records the sequence of commands performed during a game. Darmok [6] stores plans which are sequences of actions. When reusing these plans, a dependency graph between the plan’s actions is generated using the action’s preconditions and effects. Plans are captured by observing a human expert who also annotates the captured plans with applicability conditions. CBRetaliate [7] stores policies indicating the best action to take for a given state.¹

In all these situations the CBR systems follow a deterministic path; given the current state of the game once the CBR system has committed to an action/script/plan/policy, the next action/command to be executed is predetermined. This does not mean that subsequent actions/commands are also predetermined. For example, CBRetaliate might alter its current policy to improve performance. But after changing the policy, the next action is once again predetermined. The same holds true for other CBR systems playing games. For example, Darmok can adapt the current plan while executing it in the game in a process called on-line adaptation. Still, once the adaptation ends, the next action to execute is pre-determined by the plan.

In this paper we are interested in extending the representation of the observations for stochastic policies. A stochastic policy π is a mapping: $\pi: \text{State} \times \text{Actions} \rightarrow \text{Probabilities}$. For example, an agent might observe multiple games on the same map and with the same initial conditions, and at the beginning of the game it might observe players rushing to attack the opponent 30% of the time, building a townhall 50% of the time and constructing a defensive tower 20% of the time. An observer might be able to discern the difference in choices made by looking at a complete state. But the state might not be fully observable. Hence, an observer will not be able to fully understand the difference in the choices, leaving it with the need to represent the observed actions of the agent as stochastic policies.

¹ CBRetaliate stores a Q-table. This Q-table yields the greedy policy which is obtained by selecting for every state the action with the highest Q-value. The greedy policy is the solution stored in the case and the Q-table is needed for performing adaptation using Q-learning.

In this paper we study CBR systems learning from observations that can be represented as stochastic policies. While there are a number of works that combine CBR and reinforcement learning (RL) using non-stochastic policies (e.g. [7,8,9,10]), to our knowledge this is the first time a state-action probability-distribution *representation* is being used in the context of CBR systems for adversarial environments. Note that our approach does not perform RL, but is inspired by RL solution representations. This representation is more general than those discussed in Table 1 and, hence, our work advances the state of the art. Being able to reason with stochastic policies can have practical implications for CBR systems learning from observations because frequently there is no explanation for apparently conflicting decisions observed (e.g., an agent observing multiple online games from players world-wide). We describe a general framework which encompasses three steps. The agent: (1) observes agents performing actions, elicits stochastic policies representing the agents' strategies and retains these stochastic policies as cases, (2) analyzes the environment and retrieves a suitable stochastic policy, and (3) then executes the retrieved stochastic policy, which results in the agent mimicking the previously observed agents. We implement and evaluate our framework in a system called JuKeCB that observes and mimics agents playing games.

2 A Framework for Capturing and Reusing Stochastic Policies

A stochastic policy π is a mapping:

$$\pi: \text{States} \times \text{Actions} \rightarrow \text{Probabilities}$$

such that the set of a state's actions $\pi(s) = \{ \pi(s,a) \mid a \in \text{Actions} \}$ is a probability distribution for every state s in States. That is, $\sum_{a \in \text{Actions}} \pi(s,a) = 1$. Our framework captures and reuses stochastic policies. Reusing a policy means that when the agent visits a state s the agent selects an action to take based on the probability distribution $\pi(s)$. The Algorithm Reuse-StochasticPolicy below implements this.

Reuse-StochasticPolicy(π, G, P, Δ, t)

Input: π : stochastic policy; G : game engine, P : player we are controlling, Δ : time to execute policy; t : time to wait for execution of next action

Output: none

$t' \leftarrow 0$;

while not($t' \leq \Delta$) **do**

$s \leftarrow \text{currentState}(G)$

$a \leftarrow \text{select-Action}(s, \pi)$ // selects action based on probability distribution $\pi(s)$

execute(P, a) // P executes action a

wait(t)

$t' \leftarrow t' + t$

end-while

Capturing a policy means that the agent observes during each episode the states that are visited and the actions that are taken when those states are visited. Based on the actions taken for every state s , the agent elicits a probability distribution $\pi(s)$ for

each state's actions. The Algorithm Capture-StochasticPolicy below implements this. It assumes that no action/state can change more than once in one time unit.

Capture-StochasticPolicy(Actions, States, G, P, Δ)

Input: Actions: the possible actions that the agent can take; States: possible states that the agent can visit; G: game engine, P: player we are observing, Δ : time to observe

Output: π : the observed policy

```

for s  $\in$  States do
  visited(s)  $\leftarrow$  0 // a counter of how many times s is visited
  for a  $\in$  Actions do
     $\pi$ (s,a)  $\leftarrow$  0
t  $\leftarrow$  0; s  $\leftarrow$  nil; a  $\leftarrow$  nil;
while not(t  $\leq$   $\Delta$ ) do
  wait(1) // waits one time unit.
  s'  $\leftarrow$  currentState(G)
  a'  $\leftarrow$  currentAction(P)
  if (a  $\neq$  a') then
    a  $\leftarrow$  a'
     $\pi$ (s',a)  $\leftarrow$   $\pi$ (s',a) + 1
  if (s  $\neq$  s') then
    s  $\leftarrow$  s'
    visited(s)  $\leftarrow$  visited(s) + 1
     $\pi$ (s,a')  $\leftarrow$   $\pi$ (s,a') + 1
  if (a  $\neq$  a' and s  $\neq$  s') then
     $\pi$ (s,a)  $\leftarrow$   $\pi$ (s,a) - 1 // avoids double-counting
  t  $\leftarrow$  t + 1
end-while
for s  $\in$  States do
  for a  $\in$  Actions do
     $\pi$ (s,a)  $\leftarrow$   $\pi$ (s,a) / visited(s)
return  $\pi$ 

```

3 The JuKeCB Problem-Solving Architecture

We developed our ideas for stochastic policy capture and reuse in the JuKeCB architecture. The JuKeCB architecture is one that is typical of the CBR problem solving approach [11]; Fig 1 shows a high-level presentation as it applies to JuKeCB.

Our testbed is DOM, a generic domination game [7] which consists of teams of bots competing to control specific locations on a map called domination points (dom-points). The teams earn points over time for each dom-point controlled, and the first team to reach a specified amount of points wins. A team takes control of a dom-point when one or more of the team's bots is close to it without the other team being in the area for a small period of time. Bots have health points that are lost in dice-roll

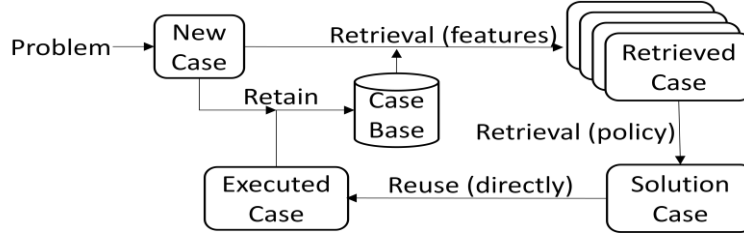


Fig. 1. The JuKeCB architecture

combat, which is initiated when bots on different teams navigate close to one-another. Combat is played to completion (only one team's bots remain), and slain bots are respawned. The dice roll is modified so that the odds of reducing the opponent health points increase with the number of friendly bots in the vicinity. The game then consists of each team trying to hold onto more dom-points than the other team.

Domination games are interesting in that individual deaths have no direct impact on the final score because kills do not give points and any player killed will respawn to continue playing. This allows for overall strategy and organization to have a far larger impact in the final outcome of the game.

Because of the large number of possible game states in DOM, we follow the state abstraction model of [7] which simply tracks ownership of dom-points, and to which dom-points bots are sent. This abstraction reduces the number of states to $d^{(t+1)}$, and the number of actions to $(b \times t)^d$ where d is the number of domination points, t the number of teams, and b the number of bots; one is added to the exponent to account for neutral ownership of dom-points at the beginning of games. We estimate the total number of possible states in the game to be at least $O(2 \times 10^{34})$, a function of: (1) the number of cells in a map of n rows and m columns, (2) the number of bots per team, (3) the remaining health, 0 to 10, of each bot, (4) the ownership of each dom-point (5) a number 0 to 5 for each dom-point, indicating the amount of game ticks since a bot began to attempt the capture; 0 is no attempt, whereas 5 transfers ownership. So the number of states is about $(n \times m)^{(b \times t)} \times 11^{(b \times t)} \times (t+1)^d \times 6^d$. In our experiments, $n = m = 70$, $b = 3$, $t = 2$, and $d = 4$. Hence, without the abstraction, it would be computationally infeasible to observe a sufficient number of actions in each state to have a representative average.

DOM showcases the need to capture and reuse stochastic policies. An agent playing a DOM game may send a bot to a third dom-point in a situation where the agent is already controlling the two other dom-points. However, later on, in the same situation, the same agent might decide to send all bots to defend dom-points it owns without sending any bot to the third locations. An observer might be able to discern the difference in choices made by looking at a complete state. But the state might not be fully observable, or, even if it is fully observable the number of features could be extremely large. In the DOM game, the state is partially observable; whereas features such as the current score and the team owner of a location is fully observable, other features such as the exact locations of the bots and their current trajectories are not. Hence, an observer will not be able to fully understand the difference in the choices, leaving it with the need to represent the observed actions of the agent as stochastic policies. In the face of this complexity we assume that the agents follow a purely

stochastic Markov strategy, in which the action to execute only depends on the current abstract state. In general, this assumption will not always hold – simple averaging can fail when an observed policy is a sophisticated one that depends on a longer history of previous states. However, we empirically found that JuKeCB performs well in DOM; *why* an enemy has chosen to use a specific strategy is not as important as *what* strategy has been observed in the case base that counters it successfully.

Each game episode is observed in real-time, and segmented into windows of observation of size Δ (a pre-determined amount of game-clock cycles). During each Δ , the features characterizing the game state are observed and recorded as the features of the new case; the policies used by each team during Δ is recorded in the case as the solution. Details of the case features and solutions are described in Section 4, as well as the process of retaining cases. Section 5 explains how and when a set of cases similar to the features of the new case is retrieved, and then details how the solutions in this set are searched for the solution to be reused. The directly executed solution is then stored as a new case in the case-base, with features appropriate for the Δ during which it was applied.

4 Case Capture and Retention through Policy Observation

Previous work on learning policies (e.g. [12], [13]) has demonstrated that state abstractions are essential for quickly learning good policies. Consequently, JuKeCB has a representation of game states (used for case features) that focuses on the most important elements for the learning task (namely the ownership of map locations, among others); as gameplay proceeds through different states, JuKeCB observes for each team the actions taken by each team member (or “bot”) in these states, and records the observations as cases. That is, over a period of time Δ , JuKeCB observes for each bot b of team t , and for all states visited during Δ , that whenever the game is in state s , bot b takes action a with probability $p_{b,t}(a, s)$. Based on this, the system builds a team policy π^t , which formally represents the *strategy* being followed by each team:

$$\pi^t: \forall \text{ BOT } b \text{ on team } t, \forall \text{ STATE } s, \forall \text{ ACTION } a, s \times a \rightarrow p_{b,t}(s,a)$$

A team’s policy π^t indicates for each game state s what is the probability $p_{b,t}(s,a)$ of a team member b taking action a in that state. We therefore define a case as a tuple: $((f_1, f_2, \dots, f_n), \pi^1, \pi^2, \dots, \pi^m)$, where:

- (f_1, f_2, \dots, f_n) are the features of the case; these abstract the game state
- $\pi^1, \pi^2, \dots, \pi^m$ are the team policies for team 1, 2, ..., m respectively

In our current testbed, we have two teams (so $m = 2$). The case features we use are: (1) the number of domination points, (2) the number of bots in each team, (3) the Manhattan distance between all domination points taking walls into consideration, (4) the percentage of time that a team held each domination point, and (5) the point difference observed between winning and losing policies. The elements which define a policy, the case solution, are the following: (1) the percentage of moves in which a

given bot went to a given domination point, (2) the percentage of moves in which a given bot went to a domination point not currently owned by its team, and (3) the percentage of moves in which a given bot went to the closest domination point. Thus a case represents a pair of stochastic policies for a single, abstracted game state.

Case Retention. Over the course of a game (JuKeCB can simultaneously play and observe; for the empirical evaluation, we do not exploit this ability) all observed cases are stored into a temporary file. When the game is over a filtering process takes place to ensure that no redundant cases are added. The filtering process goes through each new case one at a time and compares it to all cases currently stored in the case base. A new case c is added unless there is an existing case with at least 95% similar to c . If the most similar case is over 95% similar, the filtering system will then look at the Delta-score field in the two cases. If the new case has a higher Delta-score, it will replace the existing case otherwise the new case is discarded. In our experiments, JuKeCB did not suffer from dimensionality problems related to the representation of game states, and had roughly 175 cases after full training (when observing, cases are saved every 500 turns and games last about 50,000 turns).

5 Case Retrieval and Reuse

When JuKeCB is playing the game, the case retrieval process follows a 2-step retrieval process. The first step happens once before the start of the game; all cases that do not apply to the current map, as determined by the maps features, are removed from consideration. In the second step JuKeCB compares the observed game state to all remaining cases and choose the most similar one, provided that it is at least as similar as some predefined threshold. If no such a case is found, JuKeCB uses a default case, which implements an equiprobable policy (one in which all actions are equally likely to be taken).

When computing the similarity of two cases, we compute both the similarity of their features and of their stochastic policies. The similarity of features is computed by aggregating local similarities. Given two vectors of features $\langle X \rangle$ and $\langle Y \rangle$, the aggregated similarity metric is defined in the usual way:

$$\text{SIM}_{\text{FEATURES}}(X_{1..n}, Y_{1..n}) = \alpha_1 \text{sim}_1(X_1, Y_1) + \dots + \alpha_n \text{sim}_n(X_n, Y_n)$$

The sum of the vector weights, $\alpha_1 + \dots + \alpha_n$, adds to 1. As a result, $\text{SIM}_{\text{FEATURES}}()$ returns a value between 0.0 and 1.0 (1.0 being most similar). The most interesting local similarity we compute is that of location ownership; to do so, the percentage of time (ratio) that each team owned a map location is compared.

When computing the similarity of the policies, we compare a policy of the observed case with the losing policy of a case in the case base. Similarity of case solutions is computed according to the following formula:

$$\text{SIM}_{\text{sol}}(\pi^1, \pi^2) = \sum_{s \in S} \sum_{a \in A} (\alpha_{s,a} \times \text{sim}(\pi_{s,a}^1, \pi_{s,a}^2))$$

Thus, the similarity of two policies is a comparison between the frequency that action a was taken by each team in state s ($\pi_{s,a}^x$ is this frequency), for all states and actions.

The retrieval comparison $SIM(C_1, C_2)$ of a case C_2 to the case representing the current situation, C_1 , is expressed as the following weighted sum ($\alpha_{sol} + \alpha_{feature} = 1$):

$$\alpha_{sol}SIM_{sol}(policy(C_1), policy(C_2)) + \alpha_{feature}SIM_{FEATURES}(features(C_1), features(C_2))$$

Case Reuse. The strategy JuKeCB uses during gameplay is to follow the retrieved winning policy according to its probability distribution as explained in Section 4. That is, while the retrieved stochastic policy π is being executed, the selection of the next action is governed according to π by the percentage of moves in which a given bot (1) went to a given domination point, (2) went to a domination point not currently owned by its team, and (3) went to the closest domination point. The probability distribution π' recreated during case reuse is an approximation of the strategy π used by the winning team. Retrieval occurs every 525 turns (games are about 50,000 turns).

6 Empirical Evaluation

We tested the effectiveness of the JuKeCB system in the Dom game. JuKeCB observes and play games against a number of teams and build up a case base of the strategies it observes and captures. Over several experiments the JuKeCB system was tested in such a way to help determine its strengths and weaknesses.

6.1 Setup

In this section, we present the teams that JuKeCB observed, outline the training procedure for the three experiments we conducted, and the parameters used in each. In order to test JuKeCB, teams were created for it to observe and to play against. Each of these teams has a fixed gameplay strategy. Table 1 summarizes each of the fixed strategies.

The training set is slightly different for each experiment but the overall idea remains the same. In order for JuKeCB to be trained to win against a certain opponent, it must observe or by chance play a strategy which wins while the said opponent is playing. For example, if GreedyDistanceTeam plays against DomOneHuggerTeam and does well against it then JuKeCB will have received training against DomOneHuggerTeam. A full training set is one where JuKeCB observes each of the fixed strategy teams playing against every other. Ideally, after that training set JuKeCB will be able to play against any of the fixed strategy teams. For each of the tests run in the following experiments the following game variables were used: 2 teams per game, 3 bots per team, 4 domination points per map, games were until the sum of both team scores is 50,000, and each set was played 5 times (all graphs plot the mean of the set).

6.2 Experiment #1: JuKeCB vs. Fixed Teams

When JuKeCB observes a pair of teams competing against one another, then, assuming that it has observed no other teams before, it is intuitively clear that it should perform well if it plays immediately afterwards against the losing team.

Table 2. The fixed strategy teams.

Team Name	Static Policy
DomOneHuggerTeam	Sends each of its bots to Domination Point one. (Bot Destination = DomPoint 1)
FirstHalfOfDomPointsTeam	Sends bots to the first half of the Domination Points. If it has bots remaining, they patrol between the first half of the Domination Points. (Bot Destination \leq #Points/2)
SecondHalfOfDomPointsTeam	Sends bots to the second half of the Domination Points. If it has bots remaining, they patrol between the second half of the Domination Points. (Bot Destination \leq #Points/2)
GreedyDistanceTeam	Sends bots to the closest unowned Domination Point; if all are owned, goes to a random one. (Bot Destination = Closest Unowned)
Smart OpportunisticTeam	Sends one bot to every unowned Domination Point (without repeating). If not enough unowned Points, it sends multiple to the same point. (Bot Destination = Different Unowned)
EachBotToOneDomTeam	Sends each of its bots to a separate Domination Point until all Domination Points are accounted for. If it has additional bots remaining, it loops. (Bot Destination = A specific Point)

However, as it sees more and more teams and the case library grows, it is conceivable that JuKeCB's performance may degrade. The main reason is that JuKeCB is not told against whom it is playing and, hence, it needs to recognize the opponent's strategy and the situation as one similar to one encountered before. As more teams and situations are observed, conflicts may arise in the detection of the "right" opponent. We hypothesize that this will not happen. More precisely we hypothesize that the performance of JuKeCB improves as it observes more teams. For this experiment, JuKeCB will be watching and playing games against the fixed strategy teams.

The training set and testing set for this experiment is intertwined. JuKeCB team will play against every fixed strategy team in the training set before, after and during training. The following illustrates the order in which games are played:

Test: JuKeCB plays Opponent1
Train: JuKeCB watches Opponent1 play Opponent1
Test: JuKeCB plays Opponent1
Train: JuKeCB watches Opponent1 play Opponent2
Test: JuKeCB plays Opponent2

This continues until JuKeCB has played against every opponent under all possible training conditions.

On a map with four domination points JuKeCBTeam was able to beat almost every fixed policy team with relative ease. Its performance increase was measured by observing the score difference in the pre-training and post-training games. JuKeCBTeam only fails to beat one team, SmartOpportunisticTeam. It still increases

in performance considerably after having trained, however it is not enough to win. Without further training, JuKeCB is unable to counter the strategy that SmartOpportunisticTeam is able to employ. For illustration purposes, Fig 2 shows the mean score of a five game set between GreedyDistanceTeam and JuKeCBTeam. The left graph shows the results of a JuKeCBTeam which had not yet trained against GreedyDistanceTeam. In this game, GreedyDistanceTeam beat JuKeCBTeam by approximately ten thousand points. After having completed the training set, JuKeCBTeam was able to win by about three thousand points as seen in Fig 2 right. The difference displayed in both graphs is statistically significant (TTest scores: 99%). These results are typical for most of the other games ran. We repeated the experiments on a second map and the results were consistent with these results.

These results are consistent with our hypothesis that as JuKeCB sees more opponents, its performance increases. In this experiment, JuKeCB plays against every team with either an empty case base or one that is not very helpful for the match at hand. During its first match against any team, it is forced to attempt to compare it to fairly dissimilar cases or possibly to no case at all. It must use the ineffective default equiprobable policy. As such, against most teams JuKeCB fails to perform well. However once JuKeCB has been able to watch even just a few games involving the strategies that its enemy is employing its performance increases very quickly. The more cases that JuKeCB has at its disposal the better it performs.

In order to better assess our hypothesis, we tested Retaliate [13], a reinforcement learning agent, and CBRetaliate [7], an extension to Retaliate that uses CBR. Both were trained with the same sets as JuKeCB. Fig 3 shows how Retaliate (left) and CBRetaliate (right) performed after training and under the same game conditions (Number of turns in the game, number of domination points and number of bots). The abstract model makes the game look deceptively simple but beating some of the hard-coded opponents are difficult for AI agents; the Q-learning agent Retaliate and its case-based extension were beaten by the hard-coded opponent Greedy (Fig 3). In [14] an HTN planning agent was also beaten soundly by Greedy and the reactive planning approach showcased in that paper barely ties with Greedy. Given this evidence, it is remarkable how well JuKeCB did (Fig 2, right).

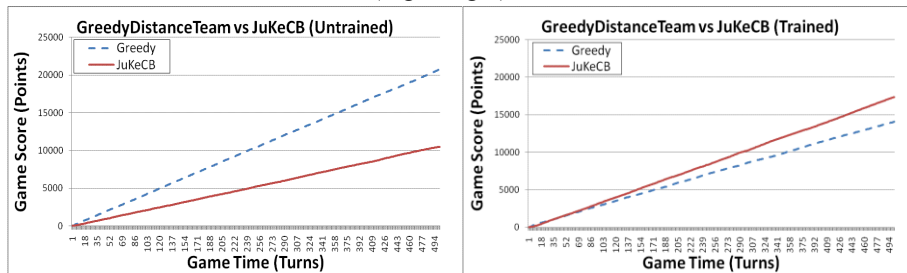


Fig. 2. Results (left) before and (right) after training versus GreedyDistanceTeam

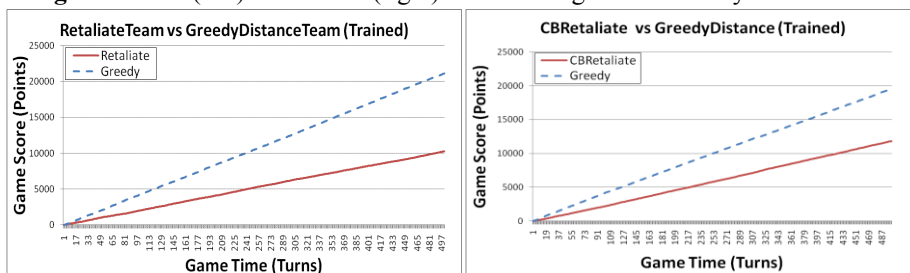


Fig. 3. Performance of Retaliate (left) and CBRetaliate (right), both after training, versus GreedyDistanceTeam

Retaliate and CBRetaliate were clearly beaten in this short game. Retaliate excels at learning another team’s strategy, however it requires a lot of game time to fully adapt to its opponent. It is much more effective at long games where its algorithm has more time to work. In this short game (50,000 turns) Retaliate does not have adequate time to learn to adapt to its opponent. CBRetaliate performed better since it can jumpstart the RL process but still was beaten.

JuKeCBTeam is not without flaws however. As the game becomes more complex, such as the games run on a map with more than four domination points, the case base becomes more complicated. There are more features which need to be taken into consideration and the amount of possible cases goes up considerably. This seems to be the main reason that JuKeCBTeam cannot beat SmartOpportunisticTeam. Since SmartOpportunisticTeam has a fairly sophisticated fixed strategy, it takes quite a few set of cases to fully define all strategies that it might employ, especially in larger maps. On the whole this experiment was a success for JuKeCBTeam. It not only showed that the underlying hypothesis was correct, but also that under certain conditions it is able to outperform reinforcement learning techniques.

6.3 Experiment #2: JuKeCB vs. DynamicTeam

In the previous experiment, JuKeCB performs well against different fixed strategy teams. Now we want to test JuKeCB versus a team that has a dynamic strategy. For this purpose we built DynamicTeam, which changes its strategy as follows:

- *First 1/3 of game:* DynamicTeam employs FirstHalfOfDomPoints Strategy
- *Second 1/3 of game:* DynamicTeam employs EachBofToOneDom Strategy
- *Final 1/3 of game:* DynamicTeam employs Greedy Strategy

Our hypothesis is that JuKeCB will quickly adapt to the changes in strategy and will continue to win by exploiting the weaknesses of each fixed strategy employed by DynamicTeam. The JuKeCB agent first plays DynamicTeam completely untrained, and slowly trains and re-plays DynamicTeam until it finally plays is with a fully trained case base.

The results were as follows: during its first match against DynamicTeam, JuKeCB handles well the first and second stage, consistently keeping a higher score than DynamicTeam. Once the Greedy strategy begins, DynamicTeam begins to pull away and eventually wins by a large margin (Fig 4, left). In the graph of the final game, JuKeCB outperforms DynamicTeam in all 3 stages (Fig 4, right).

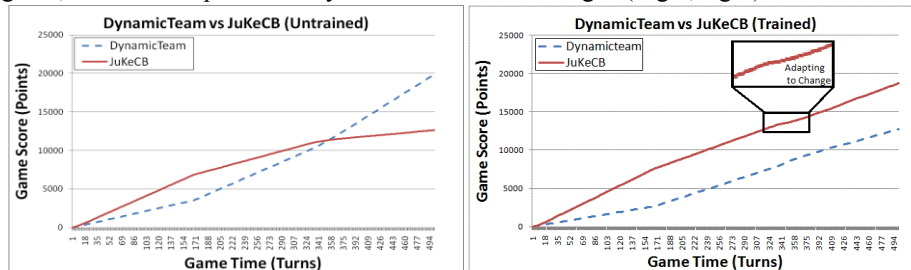


Fig. 4. Results (left) before and (right) after training versus DynamicTeam

From these two graphs we observe the occurrence of learning and adaptation. During the first match, JuKeCB performs well against DynamicTeam's first two strategies. This is expected, as the teams are static and JuKeCB can easily exploit weaknesses in their strategies. However, upon playing Greedy in the third round, JuKeCB performs poorly and ends up losing. Greedy is more dynamic and thus a more challenging opponent for JuKeCB to play for the first time. As the experiment continues and JuKeCB plays and observes more and more games, it begins to perform better; the gap between our score and DynamicTeams' score widens and becomes more dramatic over time. It quickly begins to win over DynamicTeam's first two strategies, and gradually performs better against DynamicTeams third strategy, Greedy. On the final match with DynamicTeam, JuKeCB clearly performs better against all three phases of the DynamicTeam, and wins by a very large margin. This result is statistically significant (TTest score: 99%).

Perhaps the most intriguing point made by the graphs is the quick adaptability of JuKeCB. The score lines form angles at ~ 350 and ~ 650 for both teams. This means that when DynamicTeam changed its strategy, JuKeCB immediately adapted and began to exploit the new opponent's weaknesses without delay. Not only has JuKeCB yet again proven its effectiveness as a learning agent, but it has also showcased its ability to adapt to any strategy no matter how quickly it is presented and/or changed.

6.4 Experiment #3: JuKeCB vs. Untrained Team

A trained JuKeCB team performs well against fixed strategy teams after observing them play and it doesn't perform well against them when it has not seen them. However, these experiments do not tell us how JuKeCB's ability to play against teams it has never seen before after it has observed with a large variety of other teams. We designed an experiment in which one of the team is hidden (i.e., it does not appear in the training set) and then play versus that opponent after it has trained with the full training set. For this experiment we selected Greedy because it's the hardest opponent that JuKeCB is able to beat after seen it. Our hypothesis is that JuKeCB will have a good performance versus Greedy after been subject to a large training set and despite not having been trained with Greedy itself before. Fig 5 shows the average score of the match of JuKeCB against Greedy team after it has been trained with all opponents except Greedy itself. JuKeCB's outperforms Greedy at around 130 turns and continues to widen as the game progresses. This is evidence that while JuKeCB has never seen this hidden team, it has recognized a similar situation in which a strategy performs better than the hidden team's (Greedy's) general policy.

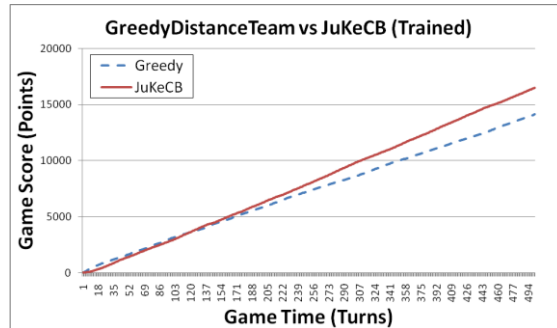


Fig. 5. Average score of JuKeCB against Greedy team after training

This experiment illustrates what is perhaps one of the most notable properties of JuKeCBR specifically, and one that has been observed with other CBR systems: their capability to generalize from the concrete instances it lazily trains from by means of exploiting similarity metrics and adaptation; in our work the concrete instances are stochastic policies. JuKeCB didn't simply mimic Greedy FixedPolicy team, nor did it guess until something worked (as a reinforcement learner would do). Instead, it looked over its case base (developed during training) and looked for previous cases that looked similar to this new team's strategy. Upon finding a good one, JuKeCB immediately uses the Case(s) and updates current ones. In other words, once JuKeCB finds a good case against this hidden team, it keeps pushing these weaknesses until they can no longer be exploited or it reaches victory.

Fig 6 combines JuKeCB's performances against the Greedy team. The 3 sloping lines represent JuKeCB's score against Greedy for that training set (untrained, hidden, and fully trained). For each of JuKeCB's lines, Greedy Team has a corresponding line in the form of $y = \{MAX_SCORE\}$ representing Greedy's final score against JuKeCB for that training set. Greedy outperforms the untrained JuKeCB agent. Upon training against all teams, including Greedy, JuKeCB performs much better than Greedy, and performs slightly better after a full training set in which it finally has seen Greedy Team play.

7 Related Work

We have discussed some related works in the introduction. Aside from those, our work is related to case-based planning [15]. Case-based planning (CBP) stores and reuses plans for solving new problems. These works frequently assume a complete domain theory is known, and hence the applicability conditions of each action in the plan can be evaluated in the current situation to determine a priori (i.e., deterministically) if it can be executed. Even CBP works that do not make the assumption about a complete domain theory (e.g., CHEF [16]) rely on the outcome of the action being deterministic. For example, CHEF knows a priori that changing one ingredient for another one "works".

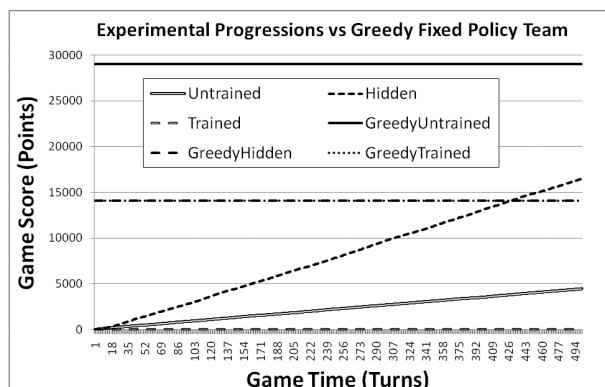


Fig. 6. Combining JuKeCB’s performances against the Greedy team

Reinforcement learning systems using techniques such as Q-learning (e.g., Retaliate [13]), can be modified to produce a stochastic policy because the underlying reinforcement learning (RL) theory is amenable to stochastic policies [12]. We do not aim at learning by trial-and-error as in RL. In fact, in our experiments JuKeCB learns stochastic policies by observing others play. Unlike RL, adapting to an opponent is not done by trying to adjust the policy but instead by reusing a different policy when the game is not going as expected. In existing approaches combining CBR and RL [8], the policies stored represent state-action values (which reflect anticipated future rewards of taking those actions) rather than observed state-action probability distributions as in our case.

8 Conclusions

We described a general framework for observing, capturing, and reusing stochastic policies. Stochastic policies are a natural representation artifact for situations in which a learning system observes an agent taking different actions when reaching the same state and the reasons behind those choices cannot be discerned. We implemented this framework in JuKeCB, a CBR system that imitates the stochastic policies it has observed in the context of a domination-style game. The efficacy of the approach was tested in three experiments. The first demonstrated that, after observing the play of teams that use a fixed strategy, JuKeCB is able to beat most of them. The second showed that, after training, JuKeCB can beat a team that changed its strategy during the same episode. The final experiment shows JuKeCB can be successful even against opponents it has never observed.

In our experiments we noted that as the number of cases stored in the case base increases significantly, the reduction in speed at which JuKeCBTeam updates its own strategy became more noticeable. Since many teams change their strategies very quickly, JuKeCBTeam must update its strategy often. Scanning through the entire case base every time a strategy change is needed can become quite costly. In the near future we will explore techniques for case base maintenance to reduce the retrieval cost of scanning the case base. For other future work, we will extend our analysis in more quantitative directions, including an analysis of different approaches to compute similarity measures between stochastic policies, such as probabilistic inference.

Acknowledgements. This work was supported in part by NSF grant 0642882.

References

1. Russell, S., and Norvig, P.: *Artificial Intelligence: A Modern Approach*, Prentice Hall, Englewood Cliffs, NJ (1995)
2. López de Mántaras, R., McSherry, D., Bridge, D., Leake, D., Smyth, B., Craw, S., Faltings, B., Maher, M., Cox, M., Forbus, K., Keane, M., Aamodt, A., and Watson, I.: Retrieval, reuse, revision, and retention in case-based reasoning. *Knowledge Engineering Review*, Vol. 20, Issue 03, September 2005. Cambridge University Press. pp 215–240. (2005)
3. Fikes, R. E., and Nilsson, N. J.: STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2, 189–205. (1971)
4. Epstein, S. L. and Shih, J.: Sequential Instance-Based Learning. In *Canadian Conference on AI*, Mercer, R. and Neufeld E. (eds.), LNCS vol. 1418, pp. 442–454 (1998)
5. Aha, D., Molineaux, M., Ponsen, M.: Learning to win: Case-based plan selection in a real-time strategy game. In: Muñoz-Ávila, H., Ricci, F. (eds.) ICCBR 2005. LNCS (LNAI), vol. 3620, pp. 5–20. Springer, Heidelberg (2005)
6. Ontañón, S., Mishra, K., Sugandh, N., Ram, A.: Case-based planning and execution for real-time strategy games. In: Weber, R.O., Richter, M.M. (eds.) ICCBR 2007. LNCS (LNAI), vol. 4626, pp. 164–178. Springer, Heidelberg (2007)
7. Auslander, B., Lee-Urban, S., Hogg, C., & Muñoz-Avila, H.: Recognizing the enemy: Combining reinforcement learning with strategy selection using case-based reasoning. *Proceedings of the Ninth European Conference on Case-Based Reasoning*, pp. 59–73. Trier, Germany: Springer. (2008)
8. Sharma, M., Holmes, M., Santamaria, J.C., Irani, A., Isbell, C., 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)*, pp. 1041–1046 (2007)
9. Bridge, D.: The virtue of reward: Performance, reinforcement and discovery in case-based reasoning. In *proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR-05)*, LNCS, vol. 3620, p. 1. Springer. (2005)
10. Molineaux, M., Aha, D.W., Sukthankar, G.: Beating the defense: Using plan recognition to inform learning agents. In the *Proceedings of the Twenty-Second International FLAIRS Conference*, pp. 257–262, AAAI Press, Sanibel Island (2009)
11. Aamodt, A. and Plaza, E.: Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Communications* 7(1), 39–59. (1994)
12. Sutton, R. S. & Barto, A. G.: *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, (1998)
13. Vasta, M., Lee-Urban S. & Munoz-Avila, H.: RETALIATE: Learning Winning Policies in First-Person Shooter Games. *Proceedings of the Seventeenth Innovative Applications of Artificial Intelligence Conference (IAAI-07)*, pp. 1801–1806. AAAI Press. (2007)
14. Munoz-Avila, H., Aha, D.W., Jaidee, U., Klenk, M., & Molineaux, M.: Applying goal directed autonomy to a team shooter game. To appear in *Proceedings of the Twenty-Third Florida Artificial Intelligence Research Society Conference*. Daytona Beach, FL: AAAI Press. (2010)
15. Muñoz-Avila, H., Cox, M.: Case-based plan adaptation: An analysis and review. *IEEE Intelligent Systems*, 23(4):75–81 (2008)
16. Hammond, K. J.: *Case-based planning: Viewing planning as a memory task*. San Diego, CA: Academic Press. (1989)