Discovering Feature Weights for Feature-based Indexing of Q-tables

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

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

Abstract. In this paper we propose an approach to address the old problem of identifying the feature conditions under which a gaming strategy can be effective. For doing this, we will build on previous work on CBRetaliate, a system that combines case-based reasoning and reinforcement learning to play team-based First Person Shooter Games. In CBRetaliate, cases are pairs (features, Q-table), where the Q-table associates a utility with each state-action pair, which is used to select an appropriate action in a given state. CBRetaliate learns cases as it plays against opponents. We propose to cluster cases in the case-base using a novel definition of similarity between their Q-tables; cases will be grouped in the same cluster if they have similar Q-tables. We propose to use standard information gain formulas and use the clusters as the classification to assign feature weights. We expect that this approach would lead to identifying features that are crucial to select which Q-table to reuse in a given situation. In addition, we propose to use the same notions of Q-table similarity to find substrategies that are common to every or nearly every case in the case base.

1 Introduction

Discovering associations between features and cases have been the subject of extensive research in the case-based reasoning literature. The INRECA project, for example, uses a variant of the ID3 algorithm [1] to build an index of the case base that retrieves cases by checking the values of the features identified to be good discriminators between the cases in a case base [2]. These kinds of approaches are typically made for classification problems, where the solution part of the case is the class under which the particular episode recorded in the case falls into. But very little such work, finding association between features and solutions of a case, has been done for synthesis problems, in which the solution part of the case is a complex structure such as a plan (i.e., a sequence of actions transforming states) or a policy (i.e., a function indicating for each possible state in the world which action to choose). Moreover, we are not aware of any work where such associations are done for Q-tables, which are functions that associate for each state-action pair (s, a) the expected utility of selecting action a in state s. Q-tables entail policies by selecting for each state the action with the highest utility and are used by reinforcement learning algorithms, which iterate by trial and error over the target domain in order to tune these Q-tables and, hence, elicit policies that maximize some reward.

In previous work [3], we developed CBRetaliate, an online learning algorithm, that uses standard reinforcement learning techniques to tune Q-tables while playing a first-person shooter game. A crucial characteristic of CBRetaliate is that it speeds-up this tuning process by reusing Q-tables stored in cases (defined as pairs (features, Q-table)). CBRetaliate is constantly monitoring the current situation (i.e., a vector of feature-value pairs) and mapping against its case base. If CBRetaliate is performing poorly in the game and there is a case that is similar to the current situation, then its Q-table is used to replace the current Q-table. Conversely, if CBRetaliate is performing well in the game, the current readings of feature-value pairs along with the current Q-table are stored in the case base. CBRetaliate demonstrated performance improvements over standard reinforcement learning [3].

We believe that there is an opportunity to exploit the knowledge learned in the case base stored in CBRetaliate. Namely, there might be similar underlying strategies for certain situations in the game and even sub-strategies that could be effective across certain states in a game. In this paper we propose a novel approach for discovering associations between features and Q-tables. We will define a notion of two Q-tables being similar and use this definition to cluster cases; each cluster will contain all cases that have similar Q-tables. We then propose to use standard information gain formulas to discover the associations between features and Q-tables and based on the results assign weights to the features. We speculate that these weights can be used to improve the accuracy of the retrieval process. Furthermore, we may be able to identify certain states (Q-table rows) in which the learned strategies are very similar across the entire case base. This could be used to improve gaming performance because we would not need to explore the search space to find good strategies for those states. Approaches such as this, where discriminants between the cases are found by looking at the cases themselves have been tried before (e.g., [4]).

The paper continues as follows. The next section gives an overview of CBRetaliate. Then Section 3 presents the definition of when two Q-tables are considered to be similar. The next section describes how clusters are built based on this similarity and how feature weights are extracted. Section 5 presents how we discover invariants, those tactics that are the same across all cases. Finally, we make some concluding remarks.

2 CBRetaliate

CBRetaliate [3] is a system that uses Case-Based Reasoning (CBR) techniques to enhance the Retaliate agent described in [5]. Retaliate uses Reinforcement Learning (RL) to play a team-based first-person shooter (TFPS) game. TFPS is a popular game genre where teams of two or more players compete to achieve some winning conditions, such as dominating special locations on a map. As a testbed for these agents, we use a configuration of a TFPS game in which individual computer-controlled players (bots) act independently but follow a team-level strategy to achieve their objectives. For the purposes of this paper, we will only summarize the essentials of those systems; please see the appropriate reference for full details.

2.1 Retaliate and Q-tables

RL is a machine learning technique wherein an agent learns a 'policy' which, for every state of the agent's world, maps an estimate of the value of taking each applicable action in that state; the goal of the agent is to maximize the sum of the rewards it receives. Retaliate uses a Q-learning variant of RL in which a policy is encoded as a 'Q-table' of expected rewards for each state-action pair.

A Q-table stores a value for each state-action pair ($Q(s,a) \rightarrow value$), where the table in this case has game states as row labels, and abstract game action names as column labels. Take for example a TFPS game where 2 teams consisting of 3 players per team compete to control 3 strategically placed locations in the game world. Because the goal of the game focuses on controlling a few map locations, it is natural to abstract the game state into tuples indicating location ownership. If 'E' is for enemy, 'F' for friendly, and 'U' for unowned, then the tuple $S = (E, F, U)$ represents the state where the enemy owns the first map location, the friendly team owns the second location, and the third location remains unowned. Similarly, because a team only gains points as a consequence of its team-members occupying the special map locations, it is sensible to abstract game actions into tuples indicating team-member destinations. So if l_1 is for 'map location 1', and so on for l_2 and l_3, then the tuple $A = (l_1, l_1, l_3)$ represents the action where players one and two are sent to map location 1, and the third player is sent to location 3. Given this model of states and actions, a Q-table in Retaliate has a total of 27 rows ranging from $S = (U, U, U)$ ... $S = (E, E, E)$, and 27 columns ranging from $A = (l_1, l_1, l_1)$ ... $A = (l_3, l_3, l_3)$, for a total of 729 state-action pairs.

At each state update from the game server, occurring roughly every four seconds, Retaliate perceives the new game state s', and the associated reward R, and uses this information to update the Q-table entry $Q(s, a)$ for the previous state s, and the action a taken in that state. The next action selected is either, with 90 percent probability, the action of highest value in the Q-table row for the associated state, or with 10 percent probability a random applicable action. By including some chance in the action selection, the agent will continue to explore new, or inaccurately valued actions. New actions are selected from the new current state, and the process continues.

The Q-table entry update calculation is performed according to the following formula, which is standard for computing Q-table entries in temporal difference learning [6]: $Q(s, a) \leftarrow Q(s, a) + \alpha(R + \gamma \times max_{a'} Q(s', a') - Q(s, a))$, which is derived from the Bellman equation [7]. 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 rate parameter, adjusts the relative influences of current and future rewards in the decision making process. Both Retaliate and CBRetaliate set γ to 1, and α to 0.2.

The reward for the new state s' is computed as the difference between the utilities of the new and previous states. 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 is computed as $R = U(s') - U(s)$.

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.

2.2 CBRetaliate as an Extension to Retaliate

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. CBRetaliate was designed to solve this problem by storing winning policies and retrieving them later based on other types of features from the game state.

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 the most similar case. CBRetaliate uses an aggregated similarity metric that combines local similarity metrics for each 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. When a case is retrieved, its associated Q-table is adapted by Retaliate by using standard RL punishment/reward action selection.

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

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

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 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. 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. In our case-base, CBRetaliate is always team A.

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 a game 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 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 Similarity Between Q-tables

Determining the appropriate feature weights requires observing which features correlate highly with the strategy selected, which requires a way to compare strategies. In the case of CBRetaliate, a strategy is represented by a Q-table, which is a matrix of expected values for each state-action pair. We suggest here two possible approaches for computing the similarities between Q-tables.

For the purpose of illustrating the trade-offs between these two approaches we present an example. Table 2 shows several small Q-tables that will be used to demonstrate the efficacy of these different similarity metrics. The Q-tables shown have 3 states and 5 actions, for a total of 15 cells. Values within the table range from -50 to 50.

		Action 0	Action 1	Action 2	Action 3	Action 4
Table A	State 0	47	-10	-23	45	15
	State 1	3	-7	15	10	-41
	State 2	37	-26	8	13	-5
Table B	State 0	20	5	15	-35	-10
	State 1	12	-32	27	-14	5
	State 2	21	30	-12	-42	17
Table C	State 0	36	-14	-30	40	11
	State 1	0	2	17	18	-20
	State 2	23	-17	12	7	-8

Table 2. The first rows of several sample Q-tables

As described in Section 2, when an agent uses a Q-table it observes the current state and by using an user-defined probability distribution either selects the action with highest value for that state or selects an action randomly from a uniform distribution, depending on whether it is emphasizing exploitation of the existing strategy (i.e., user assigns higher probability to select the action with the highest value) or exploration of alternatives (i.e., user assigns a higher probability to select an action randomly). Thus, one approach to similarity is to simply count the number of states for which the same action has the highest value. In cases where no action has a greater value than all the others, the rows could be counted as completely dissimilar (pessimistic), completely similar if the highest valued-action from the one is in the set of highest-valued actions of the other (optimistic), or as partially similar in the latter case. The idea behind this similarity is the following: Q-tables yield a policy indicating for each state, the action with the highest Q-value. This similarity is basically stating that two Q-tables are similar if their yielded policies are similar. This similarity metric is easy to compute and perfectly captures the policy or *stationary* strategy encoded in a Q-table. In Table 2, Table A has a 66% similarity to Table B and a 33% similarity to Table C, because tables A and B both have Action 0 rated highest for State 0 and Action 2 for State 1, and tables A and C both have Action 0 rated highest for State 2.

However, a Q-table used in practice by an agent using the Q-learning algorithm is not static. Rather, it continually changes as actions are rewarded or punished based on whether they lead to a more or less favorable state. Because this process may cause the highest-valued action to lose that status, the overall distribution of values throughout each row of the table is also quite important. In Table 2, Table C could be converted to something very close to Table A by a few updates, while Table B could not. In State 0, for example, both tables A and C favor actions 0 and 3 highly and consider actions 1 and 2 to be especially bad. Note that this is true even though they do not encode the same stationary strategy for this state.

The second similarity metric is designed to represent this observation. To compute this similarity metric between two tables, we average the set of cell-wise similarities between the tables. Cell-wise similarity is computed as the difference between the maximum possible distance between cell values (100 in our examples) and the absolute value of the difference between the normalized values of the two cells. The normalized value of a cell is the difference between its value and the average value across its row. This formula is written formally in Equations 1 - 3, which give the definitions of the row-wise average value, normalized cell value, and similarity metric for tables of n rows, m columns, and a maximal possible distance D.

$$\bar{A_i} = \frac{1}{m}\sum_{j=0}^{m} A_{i,j} \tag{1}$$

$$norm(A_{i,j}) = A_{i,j} - \bar{A_i} \tag{2}$$

$$QtableSimilarity(A,B) = \frac{1}{n*m}\sum_{i=0}^{n}\sum_{j=0}^{m} D - |norm(A_{i,j}) - norm(B_{i,j})| \tag{3}$$

Using this similarity metric, we find that Table A is 70.13% similar to Table B, while Table A is 94.88% similar to Table C. For an application where the Q-table is expected continue being updated, this captures our intuition about what makes one table similar to another much better than the one presented earlier.

We will adopt *QtableSimilarity* as the similarity metric between Q-tables and define that two Q-tables A and B are similar if $QtableSimilarity(A,B) \geq m$, where m is an input parameter.

4 Computing Feature Weights

Using the notion of similar Q-tables explained in the previous section we will cluster cases having similar Q-tables and use information gain formulas to extract feature weights that will serve to discriminate between the case clusters. In this section we expand on these points in detail.

Algorithm 1 shows the algorithm we present to find clusters, which is a variant of Agglomerative Hierarchical Clustering [8]. It receives as input the case base CB consisting of cases of the form (features, Q-table). It outputs a clustering

CL of the cases in CB based on their Q-table similarity. We first initialize the set of clusters making each case in CB its own cluster (Lines 1-3). The main loop will continue while no convergence point is found and CL consists of at least two clusters (Line 5). At each iteration we check if there exists a pair of distinct clusters $\{C, C'\}$ that are similar (Lines 7-8). We discuss the notion of cluster similarity in the next paragraph. If such a pair exists, clusters C and C' are removed from the set CL and replaced with a single cluster consisting of the union of the cases in both clusters (Line 9). We then break the *for* loop to restart the search for a new pair of similar clusters with the updated collection of clusters (Line 11). Once the process is finished CL is returned.

Algorithm 1 FindClusters(CB)

```
 1: $CL \leftarrow \emptyset$
 2: for each case c in CB do
 3:     $CL \leftarrow CL \cup \{\{c\}\}$
 4: convergencePoint $\leftarrow$ false
 5: while not convergencePoint and $|CL| > 1$ do
 6:     convergencePoint $\leftarrow$ true
 7:     for each pair $(C, C')$ in $CL \times CL$ do
 8:         if $C \neq C'$ and similarCluster $(C, C')$ then
 9:             $CL \leftarrow (CL \setminus \{C, C'\}) \cup \{C \cup C'\}$
10:             convergencePoint $\leftarrow$ false
11:             break
```

There are several alternatives to define cluster similarity. First, we could define two clusters to be similar if a pair of cases exists, one in each cluster, such that their Q-tables are similar. Second, two clusters could be defined to be similar if for every pair of cases, one in each cluster, their Q-tables are similar. Of these two alternatives the first one may result in fewer clusters with large number of cases because similarity relations are not transitive. A cluster may thus contain two cases whose Q-tables are very different, though both similar to a third. This will not happen with the second alternative, but it may result in too many clusters, each having very few cases. For these reasons both of these alternatives seem problematic.

Thus, we use the concept of a cluster centroid [8], which represents the entire group of elements. To compute a centroid, we simply average the contents of each cell in the normalized Q-tables of the elements in the cluster. (See Equation 2 for the definition of a normalized Q-table.) Cluster similarity may thus be computed using the Q-table similarity metric of Equation 3 on the centroids of the clusters. Clusters are considered to be similar if the similarity of their centroids is above some threshold k, and the value of k can be adjusted to control the number and size of clusters generated.

Once cases are clustered, we turn our attention to how to define the feature weights. For this purpose we will use each cluster as a unique class and calculate the information gain [9] of the various features in discriminating between the classes. Since the resulting information gains from all features add to one, we use this information gain of each feature as its weight. It is well known that the information gain is not incremental. As a significant number of new cases are added, we will need to run the clustering process and the information gain process to extract new feature weights.

Case retrieval will consider these weights in the usual manner. Namely, a case will be retrieved if the weighted sum of the local similarities is greater than a predefined threshold, as per the case similarity formula SIM in Equation 4. The weight w_i is directly the information gain for feature f_i. Hence, the sum of all weights is one. If the local similarities, $SIM_i(f_i, f'_i)$, have a value between 0 and 1 then the aggregated similarity $SIM(X, X')$ will have a value between 0 and 1. In our current implementation of CBRetaliate we assume that all weights have the same value, namely $1/n$, where n is the number of features.

$$SIM(X, X') = \sum_{i=0}^{n} w_i * SIM_i(f_i, f'_i) \tag{4}$$

5 Computing State-Action Pair Invariants

It is possible that in many games there are some actions that are either always good or always bad in a certain state. These state-action pairs often reveal themselves to expert players, who then incorporate them into their play to optimize their strategy. This is why there is a learning curve to many games as players need to start recognizing situations and learn what actions to take. One simple example of this, common to many TFPS games such as Unreal Tournament [10], is players learning that they are a harder target to hit if they constantly jump while moving. For a new player this is not an intuitive strategy, but as she plays she learns that the more she jumps the less she dies. This of course could apply to much more general and specific strategies. In the context of CBRetaliate, we believe that it is possible to find these global state-action pairs through the information encoded in the Q-tables stored in the case base. Intuitively, the granularity of the action and state definition used for the Q-tables restricts the granularity of the inferred best strategy.

As summarized in Section 2, Q-tables in CBRetaliate consist of state-action pairs where the set of states is defined as which team owns each of the domination map locations, and the set of actions indicate to which domination location each CBRetaliate team-member should go. Each row of the Q-table consists of the all the actions applicable to a given state. An example state-action pair would be a state S where all domination points are unowned, $S = (U, U, U)$, and an action A that sends each bot on the CBRetaliate team to a unique domination location, $A = (l_1, l_2, l_3)$. If A were to be executed in state S and its outcome favorable (unfavorable), then the value stored in table entry $Q(S, A)$ would be increased (decreased) so that A would be more (less) likely to be chosen in state S in the future. To extend the example, one might imagine that in all cases in a case base, the value for $Q(S, A)$ is the highest entry for state S, and can therefore be considered an invariant state-action pair. That is to say, relative to the example, every case indicates that whenever all locations are unowned, the best action is to send every bot to a unique location.

As in Section 3, we consider various alternatives to define the invariants. One possibility is to consider state-action pairs between two cases similar when, given a state (row label), the same action has the highest value in both cases. In terms of CBRetaliate, this means that some configurations of domination location ownership entail a single best assignment of team-members to locations. However, this similarity metric has the same drawbacks discussed in Section 3. Namely, that invariants would only cover fixed strategies: *if in a state S, then always go to location X*. Instead, we would like to be able to capture more dynamic strategies such as: *if in a state S, then going to either location X or Y will yield approximately the same reward* To accomplish this, we adopt a definition for state-action pair invariants based on the Q-table similarities.

Given a case base CB and a state s, we say that an invariant exists for state s in CB if for every pair of cases c and c' in CB, the rows for state s in cases c and c' are similar. We define row similarity using the same definition as Q-table similarity by viewing rows as one-dimensional Q-tables. As a result, Equation 3 is simplified to a single summation. In such a case, we define the invariant for state s to be the average of the contents of each cell in the normalized rows of the elements, just as we defined cluster centroids in Section 4.

Discovering these invariants would yield results that could be transferable to other algorithms, such as the construction of decision trees, or the creation of planner control rules in explanation-based learning as in [11]. Control rules indicate, given a state, which actions should (positive rule) or should not (negative rule) be considered. One can imagine that positive rules can be constructed from those state-action pairs of highest occurrence, and negative rules from pairs of lowest occurrence.

In the case of CBRetaliate we expect to find that, through comparing similar rows in the Q-tables, we will be able to find consistently positive and negative state-action pairs. For instance, if the enemy currently owns domination points one and two, it is not a good plan to keep all bots at domination point three since it does not improve the state, regardless of the opponent's strategy. A positive example could be if all the states are currently not owned (situation at the start of every game) it is best to send a bot to every domination point. Many other expected state-action pairs are possible, along with the potential for new and insightful strategies to emerge.

The usefulness of calculating state-action invariants is easily motivated. If enough pairs are found, a decision tree could be generated for every game situation, resulting in almost no processor power being needed for the AI. This situation is unlikely, as it would mean there is only one dominant strategy to win the entire game. Its more likely use will be in optimizing current AI algorithms. In the case of reinforcement learning, for example, the state-action pairs that are always bad can be removed from the Q-table, which would result in no waste of processing power from exploring unnecessary state-action pairs. The same is true for "bootstrapping" the exploration of good strategies; these could start off with a higher weight in the beginning end consequently ensure they are more likely to be explored first.

State-action invariants could also be used for case base maintenance. In the case of CBRetaliate, a new feature can be added that gives cases that contain Q-tables closely related to the invariants a higher weight, or remove those cases that don't contain Q-tables similar to the invariants. Application of invariants in this fashion could make it possible to further automate CBRetaliate strategy creation and storage, which could lead to even more challenging opponents.

By finding state-action pairs (un)common among all or most Q-tables it may be possible to find actions that are usually or always useful (useless). This information can then be used to create better AI in new experiments, or be used to optimize AI resources.

6 Final Remarks

Feature weighting has been the subject of extensive research in CBR (for an overview see [12]). Although most of the work have been done in the context of classification tasks, it has also been done in synthesis tasks, particularly planning [13]. In these works, the case's features are updated after the case is retrieved and reused. Depending on the outcome of reusing the case (i.e., if it yields a solution or not), feature weights are adjusted with the expectation that after a number of times that the case is reused, the case's feature weights will converge to best values in which the similarity of a new situation and the case is high if the case can be reused and low if the case cannot be reused for the situation. This tuning of case's feature weights independently of the weights of the same features in other cases is frequently referred to in the literature as *local weighting*. The approach we are proposing in this paper is complementary; initial feature weights could be obtained using the procedure presented in this paper and these local weighting techniques can be used subsequently to tune the weights. We speculate that initializing the weights in the way we presented in this paper will increase the speed of convergence to best weights.

Acknowledgments

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

References

1. Quinlan, J.R.: Induction of decision trees. Machine Learning **1**(1) (1986) 81–106
2. Althof, K.D., Auriol, E., Traphöner, R., Wess, S.: INRECA - a seamlessly integrated system based on inductive inference and case-based reasoning. In: Proceedings of the First International Conference on Case-Based Reasoning Research and Development (ICCBR-1995). (1995) 371–380

3. 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 9th European Conference on Advances in Case-Based Reasoning (ECCBR-08), Springer (2008)
4. Weber, R., Proctor, J.M., Waldstein, I., Kriete, A.: Cbr for modeling complex systems. In: Proceedings of the Sixth International Conference on Case-Based Reasoning (ICCBR-2005), Springer (2005) 625–639
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. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998)
7. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton, NJ (1957)
8. Jain, A.K., Murty, M.N., Flynn, P.J.: Data clustering: a review. ACM Comput. Surv. **31**(3) (1999) 264–323
9. Russell, S., Norvig, P.: Chapter 18: Learning From Observations. In: Artificial Intelligence: A Modern Approach. Prentice Hall (1995) 525–562
10. Epic Games, Digital Extremes: Unreal tournament. Computer Software (1999)
11. Minton, S., Carbonell, J.G.: Strategies for learning search control rules: An explanation-based approach. In: IJCAI. (1987)
12. Aha, D.W.: Feature weighting for lazy learning algorithms. In: Feature Extraction, Construction and Selection: A Data Mining Perspective, Kluwer, Norwell, MA (1998) 13–32
13. Muñoz-Avila, H., Hüllen, J.: Feature weighting by explaining case-based planning episodes. In: Proceedings of the Third European Workshop on Advances in Case-Based Reasoning (EWCBR-1996), Springer (1996) 280–294