Selecting Heterogeneous Team Players by Case-Based Reasoning: A Case Study in Robotic Soccer Simulation

Thomas Gabel Manuela Veloso

December, 2001
CMU-CS-01-165

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

It is a vital behaviour pattern of humans, the most highly developed autonomous agents, to make use of experiences accumulated in the past and to solve new problems in analogy to solutions of old, yet similar problems. This report gives an outline of our work to apply that case-based approach to an artificial agent in the domain of Robotic Soccer simulation. We enable the online coach of a robotic soccer team to determine the team line-up by a technique that incorporates knowledge into its reasoning process that was gained from former soccer matches. In order to use the knowledge contained in old cases, it is indispensable to define a meaningful evaluation of old solutions. Moreover, it is necessary to retrieve and adapt those solutions whose application to the current problem situation promises to be most auspicious. For these reasons, we also concentrate on the assessment of a team's performance. Further, we focus on a most precise calculation of the similarity between heterogeneous player types.

This research was performed while Gabel visited Carnegie Mellon from the University of Kaiserslautern, Germany. Veloso's research was sponsored by the United States Air Force under Cooperative Agreements Nos. F30602-00-2-0549 and F30602-98-2-0135.

Keywords: robotic soccer simulation, coach, heterogeneous players, case-based reasoning, genetic algorithms

Contents

1 Introduction

It is a vital behaviour pattern of humans, the most highly developed autonomous agents, to make use of experiences accumulated in the past and to solve new problems in analogy to solutions of old, yet similar problems. This report gives an outline of our work to apply that case-based approach to an artificial agent in the domain of Robotic Soccer simulation, in order to handle heterogeneous player types.

In this domain a team of agents plays against another team for a single, short (typically 10-minute) period. The RoboCup Soccer Server [4], to which all participating agents have to connect, simulates the environment. That means, it creates a virtual soccer field and provides all players with local, incomplete, and noisy perceptory information. A substantial characteristic of the Soccer Server is that it creates several *heterogeneous player types* prior to a match. These are player templates that differ in their simulated physical properties and thus in their playing skills, too.

Each team is allowed to employ a further agent, the *online coach*, which gets a noise-free, global view over the field. The online coach is intended to observe the game and to provide additional advice and information to the players. One major task for the coach is to decide which player of its team line-up is assigned to which of the available heterogeneous player types. In this work we develop an approach that accomplishes that task with the help of techniques known from the research area of case-based reasoning (CBR). Our implementation extends the online coach of Carnegie Mellon's simulated robotic soccer team ChaMeleons-01 [3].

The remainder of this report is organized as follows. We begin by discussing the basics of the problem of heterogeneous player types and the intended CBR approach. Next, we formalize and model the domain of applying heterogeneous player types in order to pave the way for the use of CBR methodologies. In order to use the knowledge contained in old cases, a meaningful evaluation of old solutions must be defined. That is why, in the following section, we deal with the assessment of the performance of a team with a particular team line-up and player types. Furthermore, given a new problem instance, it is necessary to retrieve those cases whose reutilization is feasible and advantageous. In this context a meaningful definition of similarity measures is of crucial importance because the similarity computation influences the retrieval and the adaption of cases substantially. Hence, in the next section, we concentrate on the definition and optimization of adequate similarity measures. Subsequently, we provide implementation specifics of the CBR framework we developed. The last section presents conclusion and future work.

2 Foundations

2.1 Problem Description

Each time a new robotic soccer simulation match is about to take place, the Soccer Server randomly generates a set of *player types*. Those different player types differ in their basic characteristics, such as maximum speed, size or inertia moment, depending on specified ranges for each property. Thus, the overall abilities of a single player type can vary greatly.

A team can make use of those heterogeneous player types created by the Soccer Server through the online coach that may decide which player types to use for which player. In particular it has to determine which player types are assigned to defenders, midfielders or forwards, for example[1]. Furthermore, the coach is allowed to do a limited number of substitutions during the match, i.e. changing the player type of a specific player. If there is no online coach for a team or the online coach does not deal with the application of heterogeneous player types at all, the Soccer Server will endow all players with the characteristics of a default player type.

There are several configuration parameters of the Soccer Server which affect the use of heterogeneous player types:

- team_size: number of players in a team (by default 11)
- player_types: number of player types generated by the Soccer Server (by default 7)
- subs_max: number of maximally allowed substitutions during a match (by default 3)
- pt_max: number of maximally employable players per player type (by default 3)[2]

Initially, all players are of the default player type. That is why the coach is permitted to undertake as many substitutions of player types as desired before the match starts. Obviously, a team can take advantage of the superior properties of some player types over other ones or over the default player type. Consequently, there is the need to judge which player types should be preferred to other ones and for which player roles to employ which player type.

2.2 Current Implementation

The online coach of Carnegie Mellon's simulated robotic soccer team ChaMeleons-01 consists of several modules (as shown in Figure 1), each dealing with different aspects of guiding a team through the progress of a simulated

[1]Each player may act as a goalkeeper, sweeper, defender, midfielder or forward. Throughout this text we refer to these roles as "player roles".

[2]Except for the default player type: A team is even allowed to use the default player type for all of its players.

soccer match. That modular structure simplifies the adding of new functionality and permits to change the coach's behaviour easily. Therefore new modules were added in order to solve the heterogeneous player problem CBR-based.

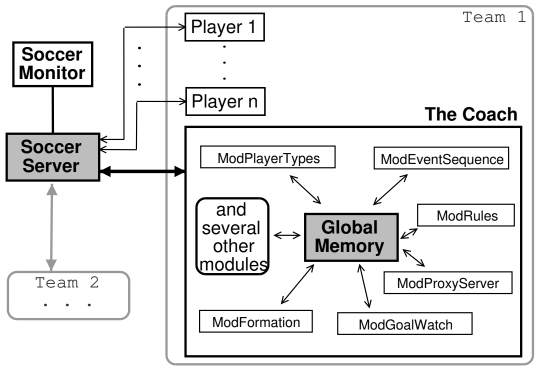

Figure 1: The role of the online coach

The previous solution to the heterogeneous player problem is realized in module `ModPlayerTypes`. It includes a rather simple evaluation of player types that basically tries to ascertain which player types are most valuable as a defender, midfielder or forward and makes use of them correspondingly. Those evaluation values of single player types were created on the basis of isolated training situations. Each type got an evaluation score for its abilities in shooting on the goal or intercepting a ball, for instance. Those evaluation scores were combined heuristically for each player role, so that a decision could be made whether a player type should be used as a defender, midfielder, or forward.

One deficiency of the named solution is represented by the fact that it does not take into account possible synergy effects that may occur due to interaction between several player types. Moreover, the continued performance of a player with its chosen player type during an actual game is not regarded.

2.3 Basic Idea for a CBR Approach

According to [14], "Case-based reasoning is reasoning by remembering." And in [17] it is stated that a "case-based reasoner solves new problems by adapting solutions that were used to solve old problems." Utilizing that basic idea to the problem of applying heterogeneous players in a robotic soccer match, old problems can be represented as the set of available player types in games of the past. Whereas old solutions are considered as the choice of player types that was made in those solutions and new problems as available player types for the upcoming match.

A main advantage of a CBR approach is the avoidance of a high effort for knowledge acquisition. Plenty of old solutions are available in the form of past games' logfiles. The experience may be exploited by a case-based reasoner.

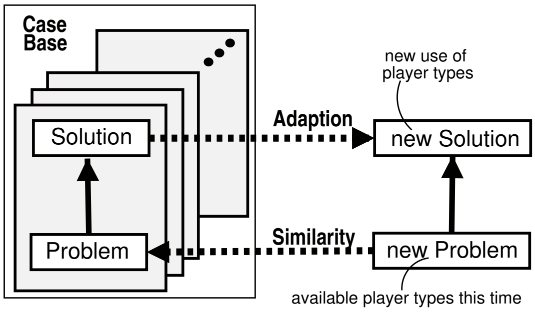

Figure 2: Using CBR for the problem of heterogeneous players

Moreover, knowledge maintenance becomes manageable easily if old problems and solutions are stored in a *case base*. For example, cases that we do not consider significant, representative, or good enough may be removed from the case base, thus excluded from the reasoning process, and substituted by other or newer cases. As a consequence, a main goal of our work was to implement a framework that analyzes existing logfiles of matches and extracts the relevant information in order to generate cases that represent a summary of a match with respect to the use of heterogeneous player types. As sketched in Figure 2, each case is supposed to correspond to a "problem" in form of available player types and to one or more "solutions" in form of taken applications of heterogeneous player types.

More details on case-based reasoning in general can be found in [2, 13, 17] and on its application to the problem in hand in Section 3.

An important concern is that the cases stored in a case base should contain some kind of assessment of their goodness. That assessment has to express how good the team performed in that case with the specific player types used. Cases with a higher evaluation value can be preferred in the retrieving phase of the CBR cycle [2]. A straightforward evaluation function might, for instance, be based just on the score of the game. Function e_1 and e_2 represent two very simple evaluation functions that are not meaningful enough.

$$e_1(x) = score_{ourTeam} - score_{theirTeam} \tag{1}$$

$$e_2(x) = \frac{score_{ourTeam}}{score_{ourTeam} + score_{theirTeam}} \tag{2}$$

We will discuss the problem of finding a more sophisticated evaluation function in more detail in Section 4.

3 Case Structure: Modeling the Domain

A major step in applying CBR to the heterogenous player problem is to construct a model of the domain and to decide on a specific structure of the cases.

In this section we first introduce the representation of a single case describing an application of heterogeneous player types. Then, we present a way to obtain significant, summarizing, and representative information on a match in the form of statistics of that match. We need this statistical information to provide a meaningful evaluation value.

In the context of case-based reasoning similarity measures are sometimes considered to be a part of the domain model. However, since we dealt with the definition of similarity metrics in very detail, we devote a separate section on that topic. In Section 5 we describe and refine an appropriate modeling of similarity measures between heterogeneous player types and between cases.

3.1 The Structure of a Case

In an object-oriented case representation a case consists of a set of objects. These objects are characterized by a fixed set of attributes, and they are composed by subsuming attributes, that belong together, to object descriptions. Each attribute may embody a more or less complex object itself.

We decided to define an object-oriented case representation for cases that depict an application of heterogeneous player types. Hence, cases in our domain consist of a multitude of objects that jointly represent a summary of a match (or a group of matches) during which certain player types were used.

In Figure 3 the overall structure of a single case is illustrated.

According to [2] a case consists of at least a problem and a solution part. Moreover, it may contain additional or alternative solutions and information on these solutions' goodness. A case in the domain presented here does contain exactly those parts plus some further information that makes it easier understandable and readable.

The Problem Part

The problem part consists of two main members. First, there are heterogeneous player types with their specific characteristics as they were created by the Soccer Server.

In version 7.x of the Soccer Server those player types differ from each other in the following eleven properties. Each of these properties is realized as an attribute of an instance of a heterogeneous player type object:

- maximal player speed (`MPS`)
- maximal stamina increment (`MSI`)
- player's decay (`PD`)
- inertia moment of a player (`IM`)
- dash power rate (`DPR`)
- player's size (`PS`)

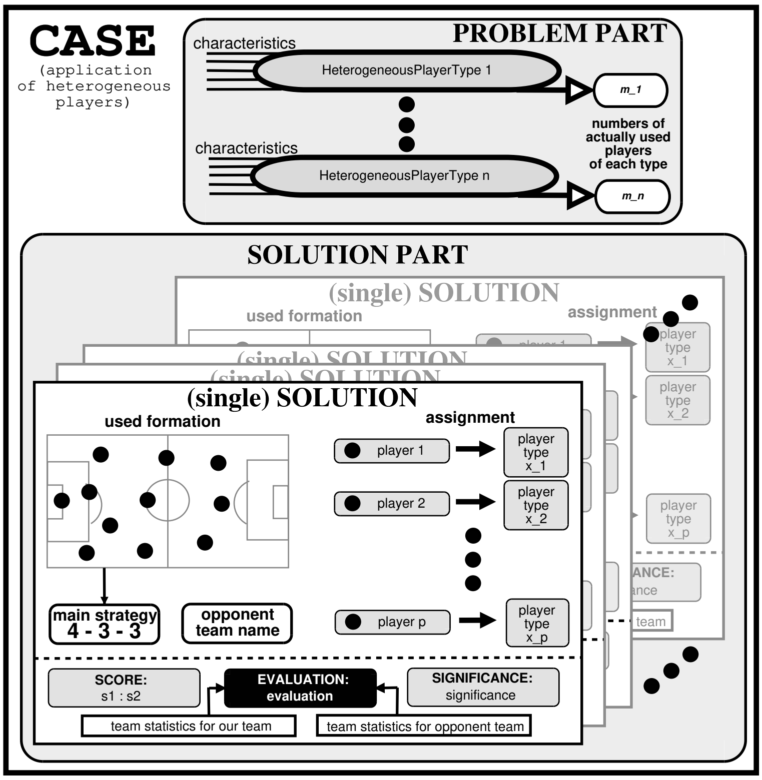

Figure 3: The case structure

- kickable margin (KM; defining the area around the player in which he has control of the ball)
- randomization of the player's kicks (KR)
- extra stamina (ES)
- maximal effort for player's actions (MAXEF)
- minimal effort for player's actions (MINEF)

For a more detailed description of those player characteristics see the Soccer Server Manual [4].

Formalizing the concept of a player type, we now give the following definitions:

Definition 1 (Heterogeneous Player Type)
Let ϱ denote the number of characteristics each player type features. A

heterogeneous player type h is defined as a vector of real numbers corresponding to its characteristics c_i:

$$h = \begin{pmatrix} c_1^h \\ \vdots \\ c_\varrho^h \end{pmatrix} \textit{ with } c_i^h \in \mathbb{R} \textit{ and } \varrho \in \mathbb{N}.$$

The set of all heterogeneous player types is defined as H.

Definition 2 (Query)
Let $Q = \{q_1, ..., q_n\}$ be a set of heterogeneous player types, i.e. $q_i \in H$ for each $i \in \{1, ..., n\}$. Then, Q is called a Query.

The other half of the problem part contains the information how many players of each player type were used. It is crucial to put that piece of information into the problem part, because it is a decisive factor for the computation of the similarity between a case and a given query[3]. To understand that, we must realize that it is the overall goal of the CBR process to make use of old cases and solutions by adapting them and applying them to the current problem.

Hence, the calculation of the similarity value between currently available player types and cases in a case base has to consider which player types were actually used in those cases. Otherwise, it may happen that the CBR retrieval picks a case out of the case base whose similarity to the given player type set mainly comes from player types that were not used at all.

Definition 3 (Problem Part p^C of a Case C)
Let τ be the number of player types generated by the Soccer Server, and let σ be the size of a soccer team. A case's problem part is defined as a matrix

$$p^C = \begin{pmatrix} h_0^{p^C} & m_0^{p^C} \\ \vdots & \vdots \\ h_{\tau-1}^{p^C} & m_{\tau-1}^{p^C} \end{pmatrix} \textit{ with } \forall i,\ h_i^{p^C} \in H,\ m_i^{p^C} \in \mathbb{N},\ \textit{and} \sum_{i=0}^{\tau-1} m_i^{p^C} = \sigma.$$

The Solution Part

As visualized in Figure 3 the solution part of a case consists of a non-empty set of single solutions. Each of those solutions may have its own characteristics; in particular those solutions can also show different kinds of applications of heterogeneous player types. When speaking about a particular solution in the course of this text, we will usually mean the actual simulated soccer match that corresponds to that solution.

[3]The similarity between a new problem and a case is always computed to the case's problem part. Furthermore, the similarity between two cases is computed between their problem parts as well.

Each solution features descriptive elements as well as evaluating attributes. Pertaining to the descriptive part of a solution, the "used formation" tells which player formation was used during that match. Formations were already part of the existing coach implementation and mainly describe the players' home regions, i.e. regions where each player is considered to stay most of the time the match. The "main strategy" is an attribute that can be chosen as a filtering parameter for the retrieval process (see Appendix C). It is inferred directly from the used formation and specifies how many players were deployed as defenders, midfielders or forwards. A further descriptive attribute is represented by the name of the opponent team.

The evaluating attributes involve the score of the match that corresponds to the solution in hand and a significance value which is proportional to the game's length. Furthermore, each solution includes a listing of values that give statistical information on a team's performance in a match, such as its ball possession time or pass success rate. Those team statistics are used to compute an evaluation value that tells how well the team performed in the current solution. In Section 3.2, we will describe the generation of those statistics and in Section 4, we show how they are used to compute a meaningful evaluation value.

Definition 4 (Single Solution s^C of a Case C)
Let Form be the set of all possible team formations, Opponents the set of all teams, and $V \subset \mathbb{R}^{|V|}$ the number of team statistics that are available on a match.

A case's single solution is defined as an 8-tuple:

$$s^C = (\ form,\ mstrat,\ assign,\ opp,\ score,\ signif,\ stat,\ eval\ )$$

with:

$form \in Form$	*the used formation*
$mstrat \in \mathbb{N}^3$	*the main strategy (numbers of defenders, midfielders, forwards)*
$assign \in [0, 1, ..., \tau]^\sigma$	*the assignment of player types, with τ as the number of available player types and σ as the team size*
$opp \in Opponents$	*the opponent team*
$score \in \mathbb{N}^2$	*the score of the match*
$signif \in \mathbb{R}$	*the significance of this solution*
$stat \in \mathbb{R}^{|V|} \times \mathbb{R}^{|V|}$	*the team statistics of both teams*
$eval \in [0, 1]$	*the solution's evaluation value.*

After having introduced several definitions, we are now also able to formalize the concept of a case in the domain of heterogeneous players:

Definition 5 (Case of Applying Heterogeneous Player Types)
A case C in the domain of heterogeneous player types is defined as a tuple of a problem part p^C and a non-empty set of solutions s_i^C:

$$C = (\ p^C,\ \{s_1^C, ..., s_l^C\}\ )\ with\ l \geq 1.$$

3.2 Team Statistics

As we already said in Section 2.3 the evaluation value of a case's solution is of high importance, since it summarizes a team's overall performance. Thus, on the one hand it indicates which assignment of player types should be preferred over other ones. And on the other hand it suggests which cases' solutions are presumed to result in a better team performance (when adapted and applied to the current problem), even if those cases' similarities to the current query are not as high. To obtain a meaningful evaluation value we rely on statistical data on the regarding match.

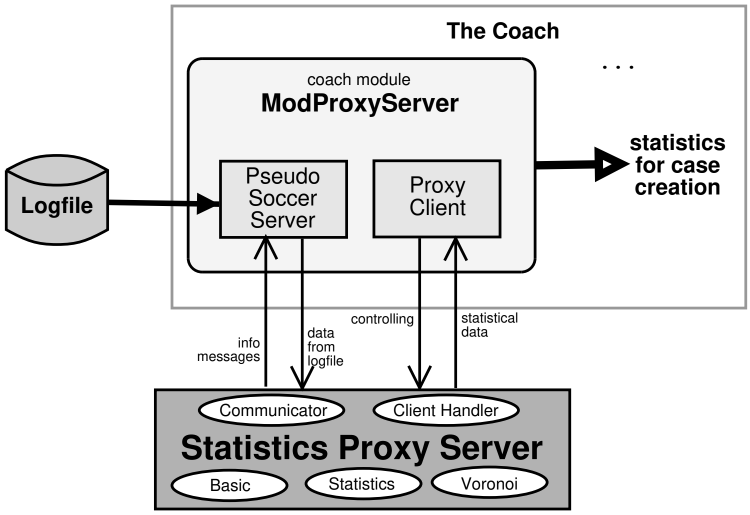

Figure 4: Using the Statistics Proxy Server to gain team statistics

For the generation of team statistics on a match we used the tool Statistics Proxy Server v1.0 [7, 8]. This client-server based software was developed to give easy access to real-time, in-depth statistics on soccer games. The Proxy Server provides statistics of three groups: team statistics, player statistics, and other, miscellaneous statistics. However, for the purpose of evaluating a team's performance we are just interested in statistical values on a team. In total, there are 33 team statistics; the complete list of those is given in the Appendix B, together with a short explanation of what each statistic means.

The proxy server is designed to generate team statistics online while a soccer match is taking place and to receive its data from the Soccer Server in real-time. Since we are interested in analyzing logfiles of past matches and in creating a case base out of them, we need an interface to the Proxy Server that imitates the run of a real match while parsing a logfile. The existing implementation of the online coach already provides such an interface. It has a module called `ModProxyServer` that establishes a pseudo soccer server which hands postprocessed information from a logfile to the proxy server as if they originated from a match taking place at the moment. Moreover, it contains an instance of a Proxy Client – a stand-alone software module that is part of the Statistics Proxy Server software package and which is designed to make the retrieval of data from the proxy server as simple as possible.

In Figure 4 we summarize the way we used the Statistics Proxy Server and the module `ModProxyServer` to generate team statistics, that we included into cases' solutions which we created from existing logfiles. Since an instance of a proxy client is part of `ModProxyServer`, we can get direct access to all the statistic values and store them for further processing.

4 Case Evaluation: Assessing a Team's Performance

The result of a match reflects only in part the performance of a team that was really achieved in that game. For example, a team may have played "really good", but lost anyway. Or it may have won by fortune, although actually having played "rather poorly". Hence, just relying on the information "score" is not sufficient for a sensible match evaluation.

We assume that each game is characterized by a "genuine result" that mirrors the actual achievements of both teams and that in general is not identical to the match's result. Thus it is our goal to find a good approximation of that "genuine result" which we intend to use to give an evaluation value on a team's performance.

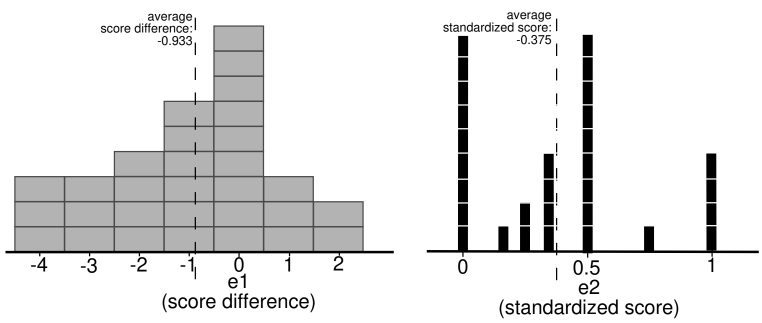

Figure 5: Distribution of match evaluation values as calculated by the simple evaluation functions e_1 and e_2

In Figure 5 we show that the use of match end scores alone is insufficient for the construction of a precise evaluation function. Both charts refer to 30 matches between two teams that used exactly the same configuration throughout that series of games. Consequently, all those games should show not too different scores and quite similar evaluation values for the performance of both teams.

The main drawback of the score-based evaluation function e_1 is that this metric is not standardized and that it is very coarse with its range of values of only 7 discrete numbers. In contrast to that, e_2 supports a continuous range of values within $[0, 1]$, but obviously a lot of match evaluation values deviate very much from their expected value $\hat{e_2} = 0.375$: About 47% of the samples (14 out of 30 matches) have an evaluation value whose distance to $\hat{e_2}$ exceeds the value of the standard deviation ($s_{e_2} = 0.326$).

Accordingly, e_1 as well as e_2 as instances of purely score-based evaluation functions are not meaningful enough. Instead of just relying on the score, we employed the 33 team statistics on a match, as created by the Statistics Proxy Server, in order to:

- construct an evaluation function that approximates the "genuine result" of a match
- assess the performance of a considered team and the goodness of its achievements as accurately as possible
- provide a numerically more stable basis for the computation of the evaluation value.

4.1 Discriminant Analysis

It is not reasonable to include the whole set V of available team statistics into the construction of an evaluation function, since that would complicate that function unnecessarily. Apart from that some of the team statistics are not at all helpful for the prediction of the game's result. In order to find out which statistics are most significant, i.e. which variables out of V contribute most to the success or failure of a team, we analyzed past soccer matches' logfiles. First, we randomly chose about 200 matches out of our logfile base. Then, we generated team statistics on those matches using the techniques described in Section 3.2. Finally, we conducted a discriminant analysis on these data sets.

Discriminant function analysis [12] is used to find out which variables discriminate between two or more naturally occurring groups[4]. That means, it dissects pre-classified data samples in order to determine and linearly combine variables to a *discriminant function* so that the group membership of new samples can be predicted as good as possible (classification task). Our pre-classified data samples are represented by the mentioned 200 past soccer matches that either were won or lost.

In the two-group case, which is relevant for our purposes, discriminant function analysis can also be thought of as (and is analogous to) multiple regression [21]; the two-group discriminant analysis is also called "Fisher linear discriminant analysis" after Fisher [6]. Computationally all of these approaches are analogous.

Definition 6 (Discriminant Function, Model)
Let $x = (x_1, ..., x_n)$ be a data sample with x_i as predicting variables and y the target variable representing the group membership of x.

[4]In our case these two groups are: matches that were won and matches that were lost. The term "variables" refers to the available team statistics.

Discriminant analysis finds a vector $\beta = (\beta_0, ..., \beta_n) \in \mathbb{R}^{n+1}$ of discriminant coefficients so that the discriminant function

$$g_\beta(x) = y = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n$$

predicts the group membership of x.

The vector β is also called model.

In our domain the predicting variables x_i are represented by the available team statistics[5]. As target variable y, which is assumed to be dependent on the statistic values, we use the final result of a match: victory or defeat of the considered team. Since the outcome variable is binary it holds that $y < 0$ corresponds to a team's victory and $y > 0$ to a defeat, respectively. The discriminant coefficients' absolute values β_i in conjunction with the variance of feature x_i give us an idea of how important the statistic x_i is for the prediction whether a team wins or loses.

For the computation of a discriminant analysis we employed the tool `logdiscr V2.0` [16]. That software performs logistic discriminant analysis whose methodological background is *Polytomous Logistic Regression* [12]. The program expects as input a set of pre-classified data samples as well as a description their format. The output of `logdiscr` consists of

- a model β containing the discriminant coefficients $\beta_1, \beta_2, ..., \beta_p$ according to Definition 6
- associated standard errors $\hat{SE}_\beta(j)$ for each discriminant coefficient β_j
- associated Wald test statistic values $W_\beta(j) = \beta_j \ / \ \hat{SE}_\beta(j)$
- associated significance values $p_\beta(j)$ (p-values)

[5]When working with the Statistics Proxy Server, we recognized several inconsistencies in the way the server generated the statistics. A first indication for one or more mistakes in the proxy server was represented by the fact that the team playing on the left side of the field played much less passes than the right team on average. On the contrary, the team playing on the right half reached a much lower pass success rate and an extraordinary high ball possession share. In order to determine whether there is a misbehaviour in the Statistics Proxy Server or not, we generated a second logfile L_2 from an existing one L_1, in which both teams had switched sides. Accordingly, the statistics generated from logfile L_1 for team A should have been exactly the same as the statistics generated from logfile L_2 for team B. But our experiment revealed considerable discrepancies so that we started searching for mistakes in the programming of the Statistics Proxy Server. On the one hand, we found that the statistics server makes the decisions which player is responsible for a movement of the ball in an unfair and unbalanced way (players of the team playing on the left side are favoured). That unfairness leads to falsified statistics on the ball possession time, the number of steals, dribblings and passes as well as on the pass success rate. On the other hand, the server includes the ball's position into the computation of the players' average x and y positions. Thus, the corresponding statistics were fudged, too. After having found and corrected the problems in the source code of the Statistics Proxy Server, we verified that it worked accurately now: The proxy server generated correct, i.e. "mirrored", team statistics for the logfiles L_1 and L_2. Both problems in the code of the Statistics Proxy Server were reported to the authors.

- for each variable $v_j \in V$ the sample mean and the sample standard deviation

The general method that `logdiscr` uses to estimate the parameters β_j is called *maximum likelihood*. In a very general sense the method of maximum likelihood yields values for the unknown parameters β_j which maximize the probability that the observed values of the outcome y deviate as little as possible from the predicted values (based upon the model β and computed by the discriminant function g_β).

In order to apply this method we need a function that expresses the probability of the observed data as a function of the unknown parameters. That function is called *likelihood function*. The *maximum likelihood estimators* of these unknown parameters are chosen to be those values that maximize this function. Thus, the resulting estimators are those which agree most closely with the observed data.

We now do not describe how to construct a likelihood function and how to infer the values for the unknown parameters. Instead, we just give a definition of that function without further explanation. For more details on this topic see [12, 21].

Definition 7 (Likelihood Function)
Let the outcome variable y be coded as 0 or 1, n be the number of samples, and p the number of predicting variables. The likelihood function is defined as

$$l(\beta) = \prod_{i=1}^{n} \pi(x_i)^{y_i} \, (1 - \pi(x_i))^{1-y_i}$$

with $\pi(x) = \frac{e^{g_\beta(x)}}{1+e^{g_\beta(x)}}$ *and g_β as discriminant function.*

However, it is easier mathematically to work with the logarithm of $l(\beta)$. The expression $L(\beta) = ln\,(l(\beta))$ is called the *log likelihood*.

4.2 Choice of Significant Variables

4.2.1 Basics on variable selection

The traditional approach to statistical model building involves seeking the most parsimonious model that still explains the data. The rationale for minimizing the number of variables in the model is that the resultant model is more likely to be numerically stable, and is more easily generalized.

In particular, in times when the outcome being studied is not so well-understood and the important variables and their associations with the outcome may not be known a *stepwise variable selection* is auspicious. Employing a stepwise selection procedure can provide a fast and effective means to screen a large number of variables. Any stepwise procedure for selection or deletion of variables from a model is based on a statistical algorithm that checks for the "importance" of variables, and either includes or excludes them on the basis of a fixed decision rule. The "importance" of a variable is defined in terms of a measure of the statistical significance of the coefficient for the variable.

In the following we make use of the *"Algorithm for Forward Selection Followed by Backward Elimination"* as introduced in [12].

The algorithm comprises a forward and a backward phase that are interconnected as illustrated in Figure 6.

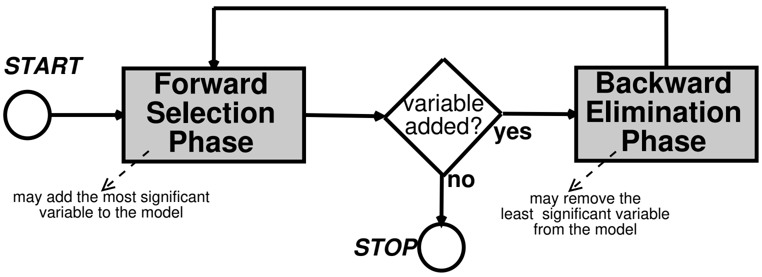

Figure 6: Schematic depiction of the "Algorithm for Forward Selection and Backward Elimination"

We summarize the modes of operation of both phases in the following:

Forward Selection Phase

Input:
- set $C \subset V$ of already chosen variables, initially empty
- set $R = V \setminus C$ of remaining variables, intially $R = V$
- significance threshold p_E for variables to *enter* the model

Output:
- variable $c \in R$ to be added to C or $STOP$

Processing:
- For each $r \in R$ fit a model β_r that includes the variables of the set $C \cup \{r\}$. Thus, we get $|R|$ different models β_r.
- Choose the model β_c with $\forall i \in R,\ L(\beta_c) \leq L(\beta_i),\ c \in R$.
- If $p_{\beta_c}(c) < p_E$, then add variable c to C and remove it from R. Otherwise: $STOP$.

Backward Elimination Phase

Input:
- set $C \subset V$ of already chosen variables
- significance threshold p_R for variables to be *removed* from the model

Output:
- variable $r \in C$ to be removed from C

Processing:
- Fit a model γ that includes all variables from set C.
- Find the variable $r \in C$ with $\forall c \in C,\ p_\gamma(r) \geq p_\gamma(c)$.
- If $p_\gamma(r) > p_R$ then exclude r from C and add it to R.

The choice of parameters p_E and p_R is crucial for the complexity of the resulting model, since it determines how many variables eventually are included in the model. The search for meaningful significance thresholds has been the focus of many research works [1, 5]. In [15] the issue of the significance level for forward stepwise logistic regression is examined. The results of this research have shown that the traditional choice of $p_E = 0.05$ can be too stringent in certain application domains, often excluding important variables from the model. Choosing a value for p_E in the range from 0.15 to 0.20 is highly recommended, in [12] a value of $p_E = 0.25$ or even larger is suggested. Whatever the choice of p_E, a variable c is judged important enough to enter a model β if its p-value $p_\beta(c)$ is less than p_E.

The second significance threshold p_R indicated some minimal level of a variable's continued contribution to the model. Whatever value we choose for p_R, it must exceed the value of p_E to guard against the possibility of having the algorithm enter and remove the same variable at successive steps. If we do not wish to exclude many variables once they have entered, then we might use $p_R = 0.9$. A more stringent value should be used, if a continued "significant contribution" is required. Anyway, if the maximal p-value, i.e. the p-value of variable r with the highest p-value, exceeds p_R, then r will be removed from the model.

4.2.2 Algorithmic realization of variable selection

We excluded the statistics on the *score*, *number of shots*, and the *shoot success rate* from the discriminant analysis since the score as well as the combination of the number of shots and the shoot success rate relate directly to the response variable y (victory or defeat of a team).

Number of observations:	218
Number of numerical attributes:	12
Number of response categories:	2
-2 LogLikelihood:	81.294035

Parameter	ESTIMATE BETA	Std.Error	Z-Value	P-Value	Sample Mean	Sample Std. Deviation
Constant	-112.2292	31.8202	-3.527	0.000		
XAverage	97.3558	37.5968	2.589	0.010	0.441	0.030
XVariance	140.2145	64.8077	2.164	0.030	0.491	0.014
YAverage	88.5449	40.9599	2.162	0.031	0.504	0.013
PassNumber	40.1776	10.4576	3.842	0.000	0.679	0.127
PassSuccess	-27.7523	15.1261	-1.835	0.067	0.570	0.066
DribbleNum	6.6747	3.4504	1.934	0.053	0.598	0.115
Compactness	101.2510	36.3555	2.785	0.005	0.495	0.031
BallPlDistAvr	89.5401	40.5247	2.210	0.027	0.514	0.019
BallPlDistVar	-291.4076	76.1467	-3.827	0.000	0.507	0.015
WinPassPattern	-11.3242	1.9162	-5.910	0.000	0.686	0.365
CornerKicks	2.0034	1.1925	1.680	0.093	0.321	0.332
KickIns	-4.5449	2.2531	-2.017	0.044	0.275	0.204

Table 1: Resulting model β_p of a discriminant analysis using stepwise variable selection

Applying the algorithm described in Section 4.2.1 to the statistical data that we generated from 200 soccer matches, we obtained the following results. Using $p_E = 0.25$ and $p_R = 0.30$ as thresholds controlling the run of the algorithm, several variables (re-)entered the model and were removed later due to p-values larger than 0.3. The algorithm terminated after 18 iterations and generated a progressive model β_p as shown in Table 1.

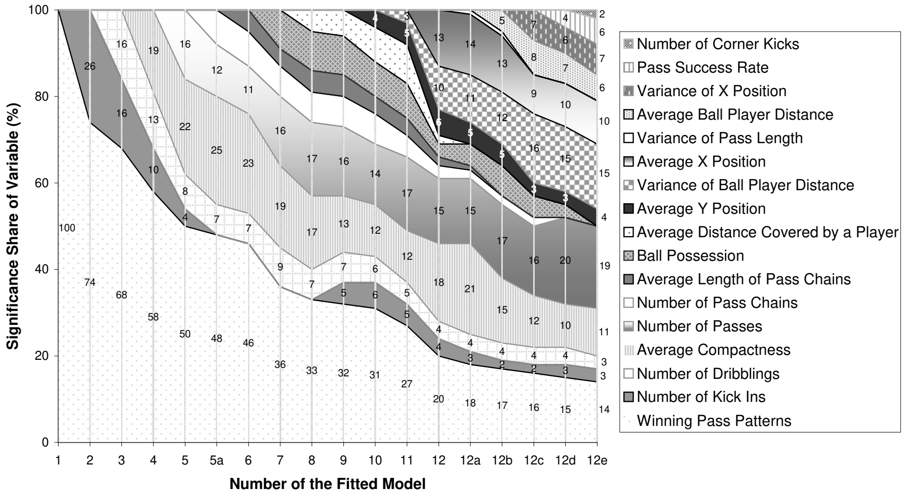

Figure 7: Stepwise variable selection with the "Algorithm for Forward Selection Followed by Backward Elimination" using parameters $p_E = 0.25$ and $p_R = 0.30$. In this figure we illustrate the relative importance of variables at successive steps of the algorithm.

It is important to mention that the absolute value of a discriminant coefficient cannot be equated with the "importance" of the corresponding variable. In fact, a statistic's influence on the model can only be estimated in correlation with the standard deviation around its mean.

In Figure 7 we give an assessment of the relative importance of statistics in a model, based on the product of the calculated discriminant coefficient with the standard deviation of the respective statistic. Furthermore, we illustrate the stepwise progress of the "Algorithm for Forward Selection Followed by Backward Elimination". We can see that, for example, in model β_7, which contains 7 variables, the statistic of winning pass patterns gets the highest importance with about 36%. However, the algorithm for stepwise variable selection considered further variables significant and added them to the model while removing others. After the final step 12e of the algorithm the average compactness of a team gets the highest importance with about 19%.

In a second application of the algorithm for stepwise variable selection we used the more conventional value $p_E = 0.10$ as a significance threshold to control the algorithm. The parameter for backward elimination p_R was set to 0.15. Using these parameters, we gained the results (model β_c) shown in Table 2.

Although all p-values of model β_p are less than 0.10 its ability to predict whether a team wins or loses, given certain team statistics on independent test data sets, is not better than the classification goodness[6] of model β_c. This is

[6]Both model, β_p as well as β_c, showed a classification error rate of about 25% on independent test data sets.

Number of observations:		218				
Number of numerical attributes:		4				
Number of response categories:		2				
-2 LogLikelihood:		113.43654				
Parameter	ESTIMATE BETA	Std.Error	Z-Value	P-Value	Sample Mean	Sample Std. Deviation
Constant	-32.4288	9.3550	-3.466	0.001		
PassNumber	9.6963	3.3867	2.833	0.005	0.6798	0.127
DribbleNum	5.5254	2.3109	2.348	0.019	0.5986	0.115
Compactness	59.4315	17.319	3.426	0.001	0.4953	0.031
WinPassPattern	-9.8190	1.4803	-6.633	0.000	0.6866	0.365

Table 2: Resulting model β_c of a discriminant analysis using stepwise variable selection

an indication that β_p may be overfitted. This means it may contain too many variables and is too specialized on the training data samples. Nevertheless, it provides interesting insights into the importance of single team statistics. Furthermore, a progressive model including more variables may be made more reliable by using more than “only” 200 training data samples. However, in the following we use the more conventional model β_c for our purposes.

4.3 Formulation of a Meaningful Evaluation Function

With means of data mining we found a classifying function in the previous section that is able to decide whether a team is likely to have won or lost, given certain team statistics on the match. Now we describe how to use that discriminator for the construction of a function that evaluates a team's performance.

Consider an arbitrary model β (as the outcome of a discrimination analysis) and the corresponding discriminating function $y = g_\beta(x) = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p$. Since we make use of team statistics x_i that are standardized to values within $[0, 1]$ and because the signs of the coefficients β_i are known we can find out the minimal and the maximal values y_{min} and y_{max} that may be assigned to the outcome variable y. As a matter of course, values near to y_{min} and y_{max} advert an obvious victory or defeat of the considered team, respectively. On the other hand values near zero indicate a rather balanced match. Accordingly, it is easy to map the value of the outcome variable y to a team performance evaluation value e_{stat} within the interval $[0, 1]$.

In Section 4.2.2 we mentioned that it is necessary to omit the team statistics that relate to shots and to the score from the process of variable selection within the scope of discriminant analysis. However, if we want to define a meaningful and comprehensive team evaluation function, we cannot completely ignore a team's abilities in shooting onto the opponent goal. Therefore, we determined to amalgamate the team evaluation value e_{stat} gained from the team statistics with a value e_{shoot} measuring a team's abilities in shooting. Thus, we allow the statistics on a team's number of shots and on a team's shoot success rate to enter the computation of the overall team performance.

A single discriminant analysis fitting a model that only contained these two

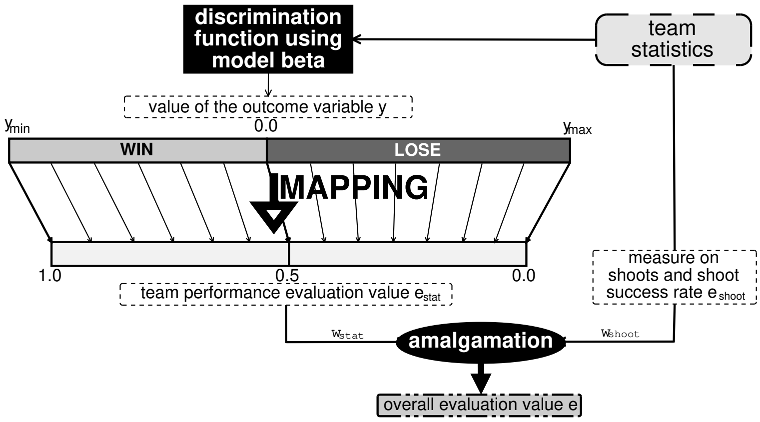

Figure 8: Mapping the discriminator's result to an team evaluation assessment

statistics revealed that the shoot success rate is about three times as valuable in predicting the match's outcome as the number of shots onto the opponent goal. Hence, we compute e_{shoot} with the equation $e_{shoot} = \frac{3}{4}x_{shootSuccessRate} + \frac{1}{4}x_{shotNum}$. The weights w_{stat} and w_{shoot} to amalgamate e_{stat} with e_{shoot} are currently chosen heuristically. We use $w_{stat} = \frac{2}{3}$ and $w_{shoot} = \frac{1}{3}$. Thus, our evaluation function e is defined as follows:

Definition 8 (Team Performance Evaluation Function e)
Let β_c denote the model according Table 2, $x = (x_1, ..., x_p)$ be a sample of statistic values on a match, w_{stat} be set to $\frac{2}{3}$, and w_{shoot} to $\frac{1}{3}$. Furthermore, let y_{min} and y_{max} denote the minimal and maximal value of function g_{β_c}. Then the team performance evaluation function is defined as follows:

$$\begin{aligned} e(x) &= w_{stat}e_{stat}(x) \ + \ w_{shoot}e_{shoot}(x) \\ &= \tfrac{2}{3}\tfrac{y_{max}-g_{\beta_c}(x)}{y_{max}-y_{min}} \ + \ \tfrac{1}{3}e_{shoot}(x) \\ &= ... \\ &= \tfrac{2}{3}(1 - 0.15x_{passNumber} - 0.08x_{dribbleNum} - 0.92x_{compactness} \\ &\quad +0.15x_{winPassPattern}) + \tfrac{1}{3}\left(0.75x_{shootSuccessRate} + 0.25x_{shotNum}\right) \end{aligned}$$

In Figure 5 we gave a frequency distribution of the purely score-based evaluation function e_2. When applied to 30 matches between the same two teams it showed an average evaluation value (standardized score) of 0.375 with a standard deviation of 0.326.

The high deviation resulting from too many extreme assessments of 0.0 and 1.0 represented one main drawback of e_2. Using our evaluation function e as given in Definition 8 for the team performance estimation for those 30 games, we obtained an reduced standard deviation of 0.129 around while the average evaluation value grew slightly to 0.431.

In Figure 9 we compare the frequency distribution of e_2 and e for the named set of matches. The chart in the lower left corner of that figure shows the frequency distribution of e_2. The second chart, however, corresponds to the distribution of e, where we grouped the actual real-numbered evaluation values for the 30 matches into categories with a width of 0.07 for the purpose of better illustration. With different shades of gray we emphasize which evaluation values assigned by e_2 correspond to values that are assigned to a certain match by e.

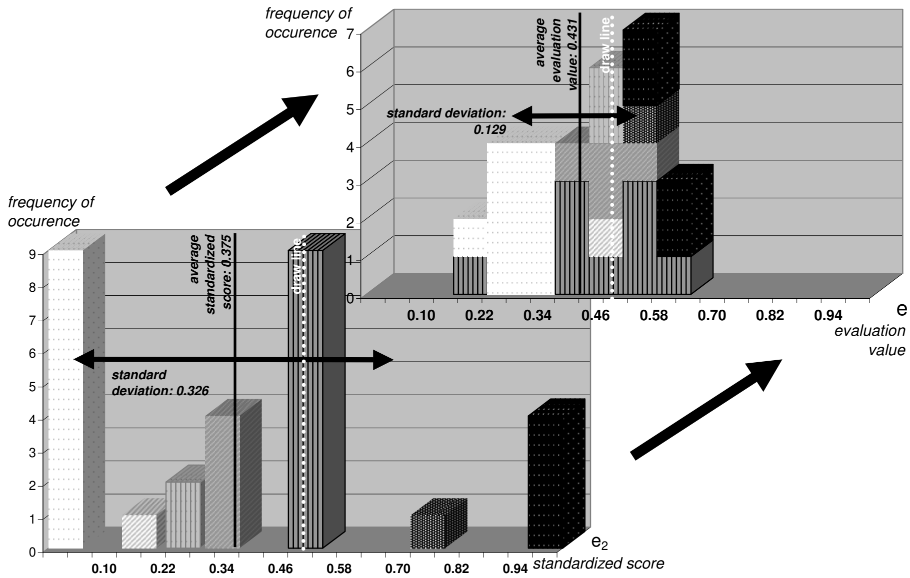

Figure 9: Comparison between the score-based evaluation function e_2 and the newly introduced evaluation function e

5 Case Retrieval: Adjusting the Similarity Measures

5.1 Basic Similarity Measures

According to [20] defining adequate similarity measures is one of the most important tasks when implementing case-based applications.

In this section we introduce similarity measures

- between heterogeneous player types
- between cases of application of heterogeneous player types

The latter similarity measure makes use of the first one, upon whose refinement we focus in Section 5.3.

5.1.1 Similarity between heterogeneous player types

The current implementation of the Soccer Server randomly influences the abilities of the player types it creates: Apart from the default player type each player type distinguishes oneself from other types by characteristics that vary from their default values. In Figure 10 we show the attribute "kickable margin" as an example illustrating that circumstance. The kickable margin represents the area around a player in which he has control of the ball. That means, if a ball is inside a player's kickable margin, he is able to pass, shoot, or intercept the ball.

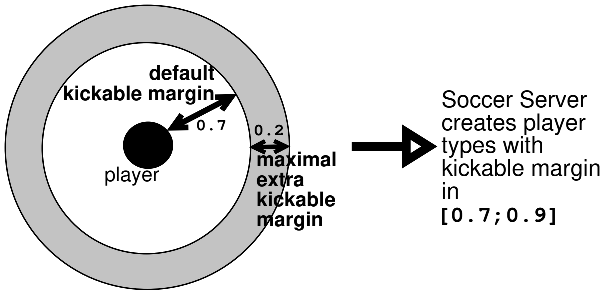

Figure 10: Different characteristics for a heterogeneous player type. Here: variability of attribute "kickable margin"

The other player characteristics are influenced randomly as well. However, the deviation from their default values may in some cases impair the player type's abilities instead of improving them. For example, the "maximum effort" of the player type's actions is decreased randomly (compared to the default value) by the Soccer Server when it creates a new player type. And a reduction of its maximal effort definitely affects a player's abilities negatively.

The flat attribute-value based implementation of heterogeneous player types suggests the utilization of a weighted similarity measure to compute the similarity between them. In so doing we first compute a similarity value for each of the characteristics, weight them appropriately, and finally amalgamate them to obtain a global similarity between heterogeneous player types. For the attribute similarities' calculation we decided to make use of a simple distance metric within the given range of the respective real-value attribute[7].

Definition 9 (Player Attribute Similarity sim_A)
Let $I_A = [min_A, max_A] \subset \mathbb{R}$ denote the interval of possible values for attribute A. Furthermore, be $v_q \in I$ and $v_c \in I$ the query's and the case's real attribute value. The player attribute similarity for attribute A is defined as follows:

$$sim_A(v_q, v_c) := \left(1 - \frac{v_q - v_c}{max_A - min_A}\right)^2$$

[7]During testing we found that squaring that distance-based similarity measure leads to more representative similarity values.

With the help of Definition 9 we define a similarity measure between heterogeneous player types:

Definition 10 (Similarity Between Heterogenous Player Types sim_{HPT}) *Let $h_1 \in H$ and $h_2 \in H$ be two heterogeneous player types. Moreover, be $(w_1, ..., w_\varrho)$ a vector of weights with $w_i \in \mathbb{R}$, where ϱ denotes the number of characteristics each player type features. Then, the similarity between h_1 and h_2 is defined as follows:*

$$sim_{HPT}(h_1, h_2) := \frac{\sum_{i=1}^{\varrho} w_i \; sim_i(c_i^{h_1}, c_i^{h_2})}{\sum_{i=1}^{\varrho} w_i}$$

The vector w of feature weights influences the overall similarity between player types on a grand scale. Accordingly, the definition of adequate feature weights is crucial for the performance of the CBR retrieval. We describe a way of finding meaningful feature weights w_i in more detail in Section 5.3.

5.1.2 Similarity between cases of application of heterogeneous player types

When defining a top-level similarity measure in the domain presented here, we have to decide what we want to connote with the similarity between a new query and an old case. In order to answer that question we have to settle how retrieved solutions are utilized.

The adaption phase of the CBR cycle comprises the adaption of an old solution to a new problem. Thus, the system has to decide which player types, that were used in that old solution can be substituted by which player types that are available at the moment. That means, the system should try to replace formerly used player types in such a manner that preferably very similar ones come into operation. Pursuing that substitution strategy, it is expectable that the new (adapted) solution matches the original one very well. While doing that adaption, the software module that makes these replacement decisions has to regard the constraint that only a certain number (Soccer Server parameter `pt_max`) of players of each type can be employed.

Function pt^* represents an operator that picks that player type out of the set of available ones which matches a given player type in the problem part of an old case best. While doing so pt^* takes into consideration that those player types which were used more frequently in the old case should be emulated as good as possible. Accordingly, those types, that were used less often, are regarded with less priority. Moreover, this operator pays attention that the maximal number of players, that can be used of a specific player type, is not exceeded.

Definition 11 (Substitutional Operator pt^*)
Let $\tau \in \mathbb{N}$ be the number of heterogeneous player types and $u_{max} \in \mathbb{N}$ the

number of players that can maximally be used of one type.

Given Q	–	*a query (a set of available player types),*
p	–	*a problem part of a case, comprising* h_i^p *and* m_i^p *for* $i \in \{0, ..., \tau - 1\}$ *according to Definition 3,*
h	–	*a player type with* $\exists i$ *with* $h = h_i^p$,

the substitutional operator pt^* *represents a* greedy algorithm *that assigns an element* $h^* \in Q$ *to any given player type* h_i^p *within the problem part* p *of a case.*

The operator pt^* *is realized as follows:*

- *be* $u \in \mathbb{N}^\tau$ *with* $\forall i \in \{0, ..., \tau\},\ u_i = 0$
- *while* $\exists k$ *with* $m_k^p > 0$
  - *choose* $j \in \{0, ..., \tau\}$ *so that* m_j^p *is maximal*
  - *choose* $h_q \in Q$ *so that*
    1. $sim_{HPT}(h_q, h_j^p)$ *is maximal and*
    2. $u_q + m_j^p < u_{max}$
  - $u_q := u_q + m_j^p$
  - *set* $m_j^p := 0$
  - *if* $h_q = h_i^p$ *then return* h_q
- *return nil*

It is obvious that it would be a mistake if the similarity measure to be defined here determined the similarity just syntactically by comparing the sets of player types that were available in the case and in the query, respectively. In fact, it is necessary to integrate knowledge about the actual use of player types into the similarity's computation. Thus, player types that were not applied in the matches corresponding to the case in hand are not to exert influence on the computation of the overall similarity value. We fulfill that requirement by using the numbers of used players of a particular player type as weights in a weighted similarity measure:

Definition 12 (Case Similarity Sim**)**
Let $Q = (h_1^Q, ..., h_\tau^Q)$ *with* $h_i^Q \in H$ *be a query and* p^C *the problem part of case* C*. Then, the similarity between query and case is defined as follows:*

$$Sim(Q, C) := \frac{1}{\sum_{i=1}^{\tau} m_i^{p^C}} \sum_{i=1}^{\tau} m_i^{p^C}\ sim_{HPT}\left(h_i^{p^C},\ pt^*\left(Q, p^C, h_i^{p^C}\right)\ \right)$$

5.2 Learning Feature Weights

5.2.1 Basics

In the last sections we have defined similarity measures that are necessary to solve the problem of heterogeneous player types with a CBR-based approach. On the one hand these metrics are sufficient to make the whole CBR framework (see Section 6) work properly. On the other hand there is still a big margin for possible improvements.

In this section we focus on improving the similarity measure between heterogeneous player types sim_{HPT} as defined in Definition 10. It is our goal to learn the feature weights w_i, that correlate to features of player types, automatically. Apart from optimizing the similarity metric sim_{HPT}, we thus get a measure for the importance of specific player attributes.

A simulated soccer match with a team playing against itself should result in a draw, provided that it is of sufficient length. Furthermore, the team statistics should converge to well-balanced values: for each team, the same number of passes, dribbles, etc. Consequently, the standardized team statistic values should gather around 0.5 and the overall team performance value e as defined in Section 4.3 will be computed to 0.5 as well. Hence, our basic assumption to realize the learning of feature weights can be summarized as follows:

Proposition 1
Let team D be a team using the default player type $d \in H$ for all of its players. Let $a, b \in H$ be heterogeneous player types with $a, b \neq d$. Further, team A is using player type a only and B is using type b only.

Let e_a be the team performance evaluation value for team A resulting from a match between A and D, and e_b be the evaluation value for team B for its match against team D.

It holds:

(a) An "ideal" match between team D and a second identical instance of team D results in an evaluation value of 0.5.

(b) If $e_a < 0.5$, the player type a is considered to be less capable than the default player type. If $e_a > 0.5$, the player type a is more capable than d.

(c) The bigger $|e_a - 0.5|$, the less similar are the player types d and a.

(d) If $e_b > e_a$, player type b is considered to be more capable than type a. Otherwise, player type a is the more capable one.

(e) The distance $|e_b - e_a|$ is an indicator for the similarity between the player types a and b.

Especially the last three parts of Proposition 1 are of crucial importance for feature weight learning for sim_{HPT}. We intend to infer the similarity between two player types a and b from the difference in performance they[8] yield in a game against a team employing the default player type only. By adjusting the feature weights w_i with a learning algorithm, we optimize the similarity metric sim_{HPT} in such a manner, that for each two player types h_1 and h_2, which yield roughly the same performance, it holds:

$$s := sim_{HPT}(h_1, h_2) \approx 1, \text{ with } s < 1 \quad (3)$$

[8]Here we mean a team that makes use of player type a or b only.

If on the other hand one of the teams obtains much better or much worse achievements than the other one, it has to hold:

$$sim_{HPT}(h_1, h_2) << 1 \tag{4}$$

5.2.2 Sample data generation

In order to fulfill equations 3 and 4 and to apply a learning technique to learn the feature weights for player type attributes a sufficient number of training data sets has to be available. Thus, we need to generate sample data on as many heterogeneous player types as possible.

As mentioned in Section 5.1.1 each player type distinguishes oneself from any other type by 11 specific characteristics. However, due to its current implementation, the Soccer Server generates the parameter values of these 11 attributes by only 5 degrees of freedom. That means, only 5 player type properties are influenced independently: Four random variables influence two attributes' deviations from their default values at a time. A fifth random variable biases the deviation of the three remaining attributes from their defaults.

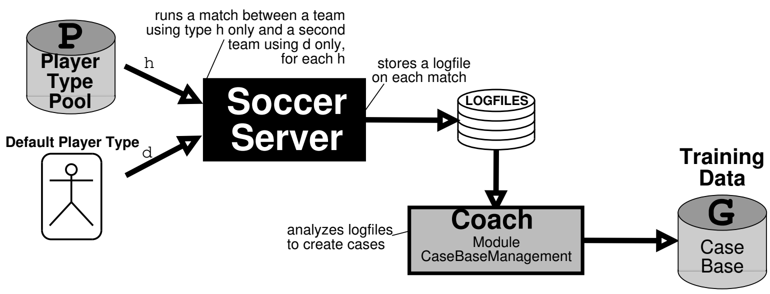

Figure 11: Proceeding to generate training data sets

Under these conditions it becomes easier to handle the whole, infinite space H of possible heterogeneous player types. Indeed, it is possible to build a *player type pool* P that covers a comprehensive subset of H and that contains a finite number of player types. In doing so, we have to make the range of each player type characteristic discrete. For the task of feature weight learning we decided that it is sufficient to discretize each attribute's range to 4 steps. Applying that kind of discretization to each independent pair or trio of player type attributes, our player type pool thus contains $4^5 = 1024$ different heterogeneous player types (number of discretization steps to the power of degrees of freedom). That way it is guaranteed that we can find a player type $h_P \in P$ for each arbitrary player type $h \in H$, so that it holds for each of its attributes A: The difference between h's value for attribute A and h_P's value for A is maximally $\frac{1}{6}$ of A's value range (worst case). On average, however, we are able to assign a player type $h_P \in P$ to any given player type $h \in H$ while the difference between each of their attribute values is only $\frac{1}{12}$ of the respective attribute's value range (average case).

Regarding the similarity between heterogeneous player types, the default player type, which is an element of the 1024 player types contained in the player type pool, can be considered as an extremum. All its attributes are set to extremal values – either at the upper end or at the lower end of the attribute's value range, depending on the particular attribute. On the other hand, however, the default player type represents a median concerning its abilities and prerequisites for performing well throughout the run of a soccer match.

With the help of the CBR framework that we developed and which is described in Section 6 we generated a *training case base* G. Each case in that base corresponds to a match between two teams T_d and T_h employing only the default player type d and the player type h, respectively. Figure 11 shows the way that this case base was created. It is important to note that the process of creating a training data case base was implemented iteratively: For each player type out of the player type pool we ran a soccer match using the Soccer Server, analyzed the resulting logfile, extracted the important information, packed them into a case and stored that case to the training data case base[9].

5.3 Similarity Teacher and Learner

5.3.1 The Learning Framework

In [20] a framework for learning similarity measures is presented. It makes use of *case order feedback* and an abstract concept, the so-called *similarity teacher* in order to realize the learning. We have adopted that approach, modified it to our needs, and applied it to learn the feature weights of the similarity measure sim_{HPT}.

We now do not want to explain the basics and foundations of the mentioned approach. For more details we refer to the paper "Learning Feature Weights from Case Order Feedback" [20]. Instead we focus on our modifications of that learning framework.

The learning approach's starting point is represented by the sample data whose generation we described in the previous section. We now presume the existence of two case bases of identical size: the player type pool P and a base of sample games G. Each of those games corresponds to one element $h \in P$ and represents a match between a team using player type h only and a second team using just the default player type.

Since it is our goal to optimize the retrieval of heterogeneous player types, we are mainly interested in the result of a retrieval, i.e. the similarity computation between a particular player type as query and a set of player types in a case

[9]Due to computational limitations we could not create a case for each player type in the player type pool. Our training data case base contained 100 cases appertaining to 100 out of the 1024 player types in the player type pool. We stress that the learning results (see Section 5.4) may be improved by increasing the number of training data sets.

base of heterogeneous player types. For that reason we give a definition for the *result of a player type retrieval*:

Definition 13 (Result of Player Type Retrieval)
Given a case base $P = (h_1, ..., h_s) \subset H$ of heterogeneous player types, a player type query $q \in H$, and the similarity measure between heterogeneous player types sim_{HPT}. The corresponding retrieval result is defined as a vector of player types:

$$H^{sim_{HPT}}(q, P) = (r_1, ..., r_s)$$

where $\forall i, j$ *with* $1 \leq i < j \leq s,\ r_i \in P$ *and* $sim_{HPT}(q, r_i) \geq sim_{HPT}(q, r_j)$.

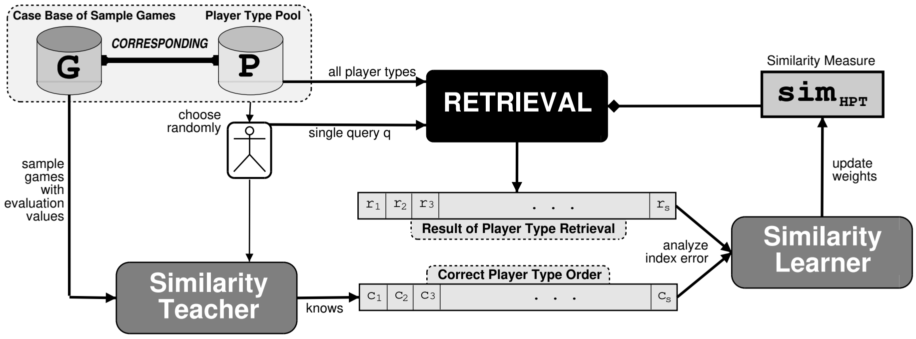

Figure 12: Framework for Learning the Feature Weights of sim_{HPT}

The case order in a result of a player type retrieval is determined by the similarity measure sim_{HPT}. But because we proceed on the assumption that sim_{HPT} is imperfect and needs to be adjusted, we can conclude that $H^{sim_{HPT}}(q, P)$ is faulty as well. The correct case order, however, i.e. the correct order of player types for a particular query, is known by the so-called *similarity teacher*. We define the teacher's teaching paradigm, the *correct player type order*, as follows:

Definition 14 (Correct Player Type Order)
Let $x_{h,d}$ be the tuple of team statistics on Team A resulting from a match where Team A employed player type h only, while Team B used the default player type d only. Then the function

$$\begin{aligned} e_d : H &\rightarrow [0, 1] \\ h &\mapsto e(x_{h,d}) \end{aligned}$$

assesses the performance of a player type $h \in H$ compared to the default player type. Further, let $P = (h_1, ..., h_s) \subset H$ be a case base of heterogeneous player types and $q \in H$ a single player type that is used as query. The correct player type order is defined as a vector of player types:

$$H^{correct}(q, P) = (c_1, ..., c_s)$$

where $\forall i \in \{1, ..., s\},\ c_i \in P$. *And it holds*

$$\forall i, j \ with \ 1 \leq i < j \leq s, \ \ |e_d(c_i) - e_d(q)| \ \leq \ |e_d(c_j) - e_d(q)| \qquad (5)$$

Hence, the similarity teacher defines the correct player type order subject to the distance of the evaluation values of query q and other player types in P. In the following we assume that the relation given in 5 is transitive.

The goal of our similarity learner is to optimize the sim_{HPT} in such a way that the case order $H^{sim_{HPT}}(q, P)$ in the result of a player type retrieval matches the correct player type order $H^{correct}(q, P)$ given by the teacher as good as possible.

In order to measure the difference between $H^{sim_{HPT}}$ and $H^{correct}$, we have to define an appropriate error function.

Definition 15 (Index Error)
Consider $q \in H$ as query, a case base of player types P, and sim_{HPT} as similarity measure between heterogeneous player types. The index error between a result of a player type retrieval and the corresponding correct player type order is defined as follows:

$$\begin{aligned} E_{index}(q) \ = \ & E_{index} \ (\ H^{sim_{HPT}}(q, P), \ H^{correct}(q, P) \ ) \\ := & \sum_{i=1}^{s} \ | \ i \ - \ indexOf(\ r_i, \ H^{correct}(q, P) \ ) \ |^{\alpha} \end{aligned}$$

where $\quad indexOf : H \times H^s \rightarrow \mathbb{R}$

$$(h_{ref}, \ (h_1, ..., h_s)) \mapsto \begin{cases} i & if \ h_{ref} = h_i \\ \uparrow & else \end{cases}$$

returns the position of a player type within a vector of player types.

The index error computes the total deviation of the elements in a result of a player type retrieval from their positions in the correct player type order. These deviations are squared[10] in order to stress that bigger deviations of elements from their correct position are much worse than smaller ones. Hence, the index error can be understood as a measure for the "disorder" in the result of a player type retrieval $H^{sim_{HPT}}$ compared to the correct player type order $H^{correct}$.

If it is possible to adjust the weights of sim_{HPT} in such a way that $E_{index}(H^{sim_{HPT}}(q, P), H^{correct}(q, P)) = 0$ for a given query q, then our overall learning goal is reached – but only for that specific query q. However, we intend to optimize sim_{HPT} in a more global manner, i.e. we want to improve (minimize) E_{index} for any arbitrary $h \in H$. To reach that goal we need the following definitions:

Definition 16 (Training Example, Training Data)
Let $q \in H$ be a heterogeneous player type, called a query. Moreover,

[10] We set $\alpha = 2$ in the following.

$H^{sim_{HPT}}(q, P)$ be the affiliated result of a player type retrieval and $H^{correct}(q, P)$ be the corresponding correct player type order. Then, the tuple

$$T(q) \;=\; (\; q,\; H^{sim_{HPT}}(q,P),\; H^{correct}(q,P)\;)$$

is called a training example.

Let $Q_s = \{h_1, ..., h_s\} \subset H$ be a set of heterogeneous player types, then

$$T_{Q_s} \;=\; \{\; T(h_1),\; ...,\; T(h_s)\; \}$$

is called training data.

Definition 17 (Average Index Error)
Given a training data T_{Q_s} on query set $Q_s = \{h_1, ..., h_s\} \subset H$, the average index error for T_{Q_s} is defined as:

$$\tilde{E}_{index}(T_{Q_s}) \;=\; \frac{1}{s}\sum_{i=1}^{s} E_{index}(\; H^{sim_{HPT}}(h_i, P),\; H^{correct}(h_i, P)\;)$$

After having defined the average index error, we now can concretize our learning goal: If we have a training data T available, we will try to find a similarity measure sim_{HPT} so that $\tilde{E}_{index}(T)$ is minimized. That optimized similarity measure contains weights which cause a result of a player type retrieval to be very close to the respective correct player type order.

In Figure 12 we give an overview of the learning framework described so far.

5.3.2 The Learner: Usage of Genetic Algorithms

The crucial module within our learning framework is represented by the learner. Its task is to minimize the average index error $\tilde{E}_{index}(T_{Q_s}$ for a given query set. The starting situation for the learner can be characterized by the following available input:

- case base of player types: the player type pool P
- a query set $Q_s = \{h_1, ..., h_s\} \subset H$ with $\forall i,\; h_i \in P$
- similarity measure sim^{w}_{HPT} with an initial weight vector w
- a similarity teacher, providing the correct player type order and the average index error computation (see Section 5.3.1)

We make use of a *genetic algorithm (GA)* that performs an evolutionary search for a minimum of the error function $\tilde{E}_{index}$. Genetic algorithms are search algorithms based on the mechanics of natural selection, natural genetics, and the basic principle of the "survival of the fittest". According to [9], they combine survival of the fittest among string or bit structures (called the *genome*) with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search.

In every generation, a new set of artificial creatures (individuals) is created using pieces (genes) of the genome of individuals of the previous generation. Occasionally, mutations are introduced, sometimes leading to improved fitness of the offspring. The general evolutionary reproduction cycle can be illustrated as in Figure 13.

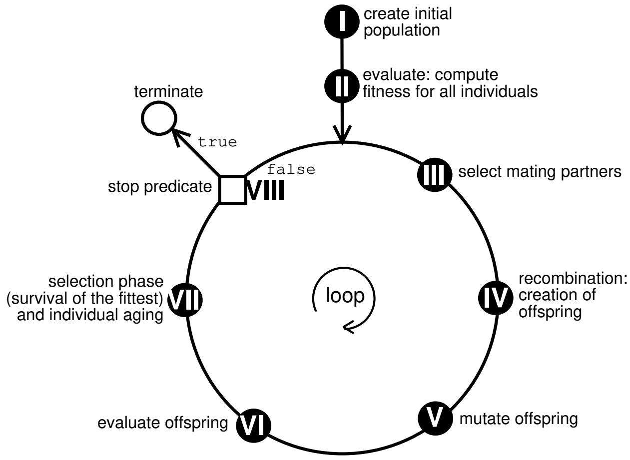

Figure 13: Generational loop for a GA

Genetic algorithms are theoretically and empirically proven to provide robust search in complex spaces, since they are computationally simple, yet powerful in their search for improvement [11].

In the following, we describe how we realized the eight fundamental steps towards an implementation of a GA searching for optimized feature weights in the similarity measure sim_{HPT}.

I Creation of the initial population Before creating a population of individuals, we have to settle how to represent a single individual and its genome, respectively: An individual corresponds to a weight vector w as used in the similarity measure sim_{HPT}, and each chromosome represents one single weight. The chromosomes are implemented as bit strings that are interpreted as natural numbers, so that the value range for the weight w_i is made discrete[11].

The population our learner uses is of constant size μ. At the beginning of the learning process, the learning module generates μ individuals, where the chromosomes (and thus the initial weight weight values) of a certain number of individuals are generated randomly. The remaining individuals are initialized to a uniform weight vector, that means to equal chromosomes.

[11]The number of bits per chromosome can be specified, of course. For most of our experiments we encoded each weight with 5 bits so that an individual with its 11 weights/chromosomes consists of 55-bit genome.

No.	MPS	MSI	PD	IM	DPR	PS	KM	...	fitness/$\tilde{E}_{index}$
	...		...		...		...		
3	01101	10111	00100	10001	01110	11000	10000	...	1532,4
4	00111	00111	00111	00111	00111	00111	00111	...	1520,7
5	00101	01011	01101	00111	10010	01011	01111	...	1592,8
	...		...		...		...		

Table 3: Exemplary representation of three individuals as used by the GA

II Evaluation: Fitness computation for a single individual The fitness of an individual ι is equal to the average index error $\tilde{E}_{index}$ that results from a retrieval with a similarity measure $sim^{w_\iota}_{HPT}$ that uses the weight vector w_ι that ι represents. Hence, an individual's fitness computation comprises:

- the determination of the retrieval result with the similarity measure $sim^{w_\iota}_{HPT}$ for each query in the query set Q_s
- the computation of the corresponding player type orders and respective index errors $E_{index}(q)$
- the calculation of the average index error $\tilde{E}_{index}(Q_s)$

III Selection of mating partners In [19] it is stated that it is misleading to define "fitness" in a sense that selection of the fittest is modeled as "mating selection" only. That would mean, that fitter individuals produce more offspring than the less fit ones.

Our learner randomly chooses the mating partners out of the current population, where always exactly *two* parents are involved in reproduction.

IV Recombination to create offspring Our recombination operator realizes a *uniform crossover*. That means, for each chromosome w_i^{child} of the future offspring the recombination operator chooses randomly either the first or the second parent's chromosome ($w_i^{parent_j}$ with $j = 1, 2$).

V Mutation The mutation operator in our genetic algorithm weight learner is implemented in a rather simple way and mainly depends on a globally defined mutation rate σ, which is adapted dynamically depending on the progress of the evolution. That operator modifies a recombined individual in such a way that each bit within the individual's genome is flipped with a probability of σ.

VI Offspring evaluation The offspring's evaluation is done in the same way as the evaluation of individuals created for the initial population (see II).

VII Selection phase Our selection operator follows a $(\mu, \kappa, \lambda, \rho)$-strategy [19], which is an extension of the $(\mu + \lambda)$-strategy [18]. The symbol μ denotes the number of individuals (and thus possible parents), appearing at a time in a population, and can be equated with the size of the population. The symbol λ stands for the number of all offspring created within one (synchronized) generation. In the $(\mu + \lambda)$-strategy the λ offspring and the μ individuals of the preceeding generation are united, before the μ fittest individuals are selected from this set of size $\mu + \lambda$.

The $\mu + \lambda$-strategy implies that each parent may live eternally, if no child achieves a better or at least the same quality. The $(\mu, \kappa, \lambda, \rho)$-strategy introduces a maximal life span of $\kappa \geq 1$ reproduction cycles (iterations) for the individuals. Moreover, that strategy allows a free number ρ of parents to be involved in a reproduction. As already mentioned in III, we chose $\rho = 2$ for our experiments, as it is typical for genetic algorithms in general.

Since each individual "dies" after κ iterations, a very good vector of weights may go lost. For that reason, we store the weights of the fittest individual of all times continuously. That way we ensure that the optimum found so far is at hand, when the stop predicate becomes true.

VIII Stop predicate The stop predicate may become true, either if the fitness value falls below a certain threshold or if a certain number of generations has been reached. We implemented the stop criterion according to the latter option.

5.4 Experimental Results

We applied the learning framework introduced so far to random query sets Q_s with size $s = 3, 5$, and 10. Since we intended to obtain statistically significant results, we repeated the learning experiment 200 times for each choice of s.

The GA learner tried to adjust the feature weights of sim_{HPT} so that the average index error $\tilde{E}_{index}(Q_s)$ was minimized. Here, we decided to let the genetic algorithm terminate after 1000 generations. The results revealed that almost no further improvement of the index error would have been expectable, if we had increased the number of generations. That is to say, the population of weight vectors reached its maximal fitness within the first 1000 generations.

A first insight we gained was that the error function's run within the 11-dimensional search space of feature weights can informally be characterized as "rather flat". That means, there are no "extreme" maxima or minima. Nevertheless, the learner manages to find a minimum of the average index error function where the error value is between 50% and 75% of the initial index error's value.

Using randomized initial weight vectors would have improved the initial average index error slightly, but would neither have had an important impact on the learning process nor on the learning results. Since the development of the actual weight values by time can be illustrated much better when using the same initial values for all weights, we determined to always start the learning process with a uniform weight vector.

In Figure 14 we show the development of the average index error subject to the number of evolutionary generations. The values shown in that chart are averaged ones, averaged over the 200 times we repeated the experiments. Obviously, the index error subsides the longer the genetic algorithm runs, converging to a minimum depending on the size of s. The improvement share is the better, the smaller the size of the respective query sets. But even for query sets comprising 10 randomly chosen heterogeneous player types the learner succeeds in improving the initial average index error, reducing it to about 70% of its starting value.

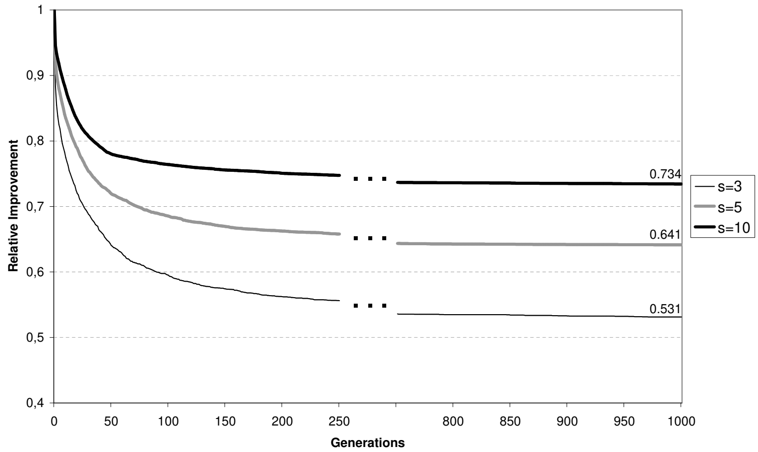

Figure 14: Relative improvement of the average index error for different query set sizes

A further remarkable observation regards the absolute values of the learned weights for the player type attributes. As already mentioned in Section 5.2 the 11 characteristics of player types are created by only 5 degrees of freedom (due to the specifics of the Soccer Server's implementation). This fact is reflected in the learning result, namely in the learned weights: There are five groups (four pairs and one triple) of attributes, whose weights are approximately identical. The attributes within one group, however, are exactly those, whose values during player type generation are determined with the help of the same random variable, i.e. they belong to the same degree of freedom.

In Figure 15 the percental weights for the 11 player type properties are given, where the abovementioned grouping of attributes is intelligibly discernible. The depiction in that chart represents the learning result for $s = 10$ after 1000 evolutionary generations of the GA learner. According to this, the most important player characteristics are the maximal player speed and the maximal stamina increment, whereas the dash power rate and the player size are relatively insignificant.

For the creation of a player type pool (see Section 5.2.2) the fact that the heterogeneous player types are generated by 5 degrees of freedom only proved to be advantageous, since that way the space of possible player types is confined.

However, the reduced number of degrees of freedom affects our leaning technique prejudicially. Due to this, the 11 features of a player type cannot be treated independent from each other. On the basis of our learning procedure's results, we can at most anticipate which of the two (or three) attributes within one group truly features the higher significance. For instance, we may surmise that the dash power rate has a higher significance for a player type's abilities than its "fellow attribute" player size because of its noticeably higher weight. Further, we may presume the same for the kickable margin and the randomization of kicks. Yet, the method by which the Soccer Server generates new heterogeneous player types impedes a more accurate analysis.

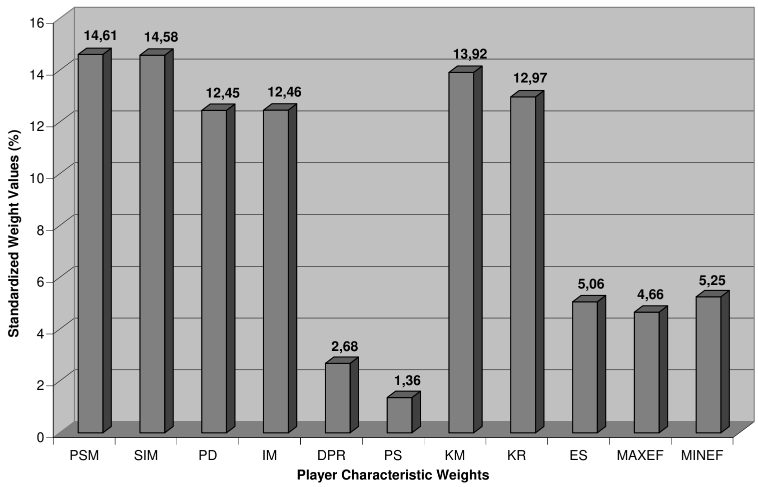

Figure 15: Learning result for $s = 10$; the final weights for the 11 player type attributes after 1000 life cycles of the GA weight learner

But analyzing the importance of player characteristics was intrinsically only our secondary concern. Our main learning goal was to optimize the feature weights of the similarity measure sim_{HPT} – and the learned weights represent an optimization of that measure indeed, as shown in Figure 14.

For the reasons mentioned above, the 11 player type properties should actually be divided into 5 not interdepending groups:

- maximal player speed and maximal stamina increment (`MPS/MSI`)
- player's decay and inertia moment of a player (`PD/IM`)
- dash power rate and player's size (`DPR/PS`)
- kickable margin and randomization of the player's kicks (`KM/KR`)
- extra stamina, maximal effort for player's actions, and minimal effort for player's actions (`ES/MAXEF/MINEF`)

We illustrate the development of the standardized feature weights during learning with our framework in Figure 16. In that picture the percental weights of the 5 named attribute groups are shown, subject to the evolutionary run of the genetic algorithm and averaged over the 200 times we repeated the experiment for $s = 10$.

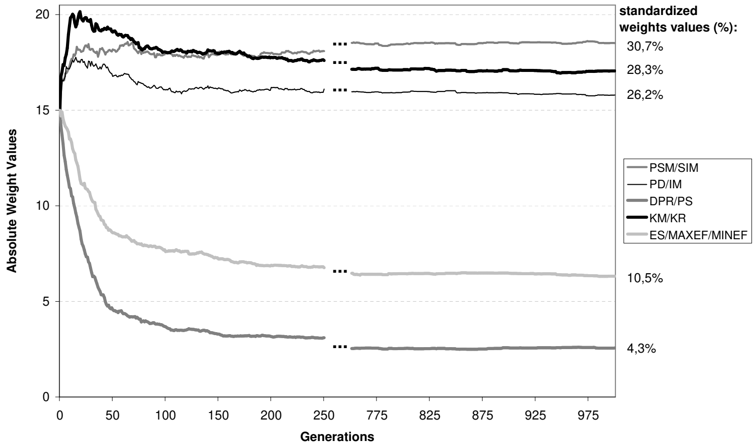

Figure 16: Progress of learning – feature weights for the 5 attribute groups

6 Implementation of the CBR Framework

6.1 The Overall Class Structure

In Figure 1 we illustrated the overall structure of the online coach of Carnegie Mellon's simulated robotic soccer team and its composition of modules. For the purposes of the present work we developed two new modules, that are realized as C++ classes:

- module `ModCaseBaseManagement`
- module `ModCBRHeterogeneousPlayers`

Moreover, we packed all the necessary functionality to provide CBR mechanisms, such as query creation, retrieval, case base handling, case adaption and so forth into an auxiliary module `CBRSupport`, which cannot be considered as a proper coach module. Instead, both new coach modules mentioned above make use of the CBR functionality realized in `CBRSupport`. In Figure 17 we give an overview of the class structure that realizes the named three modules.

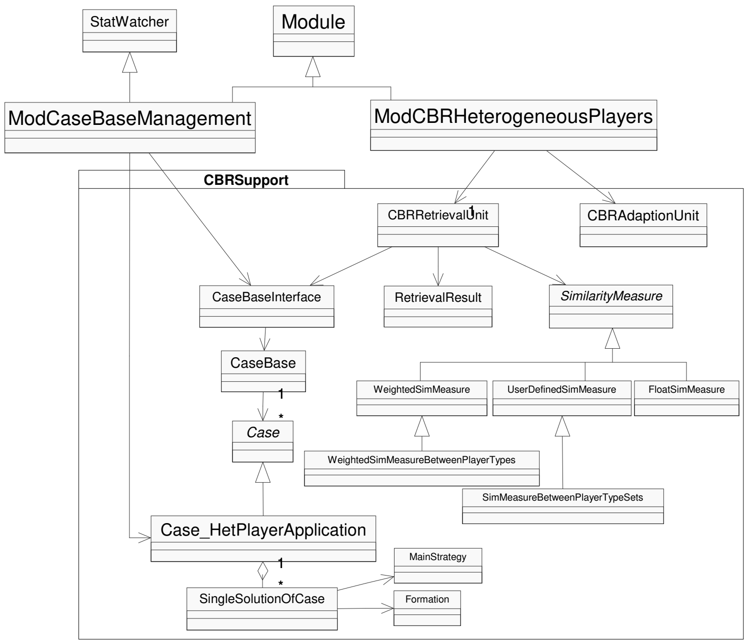

Figure 17: Class diagram for the CBR framework

6.2 Case-based Reasoning Routines (CBRSupport)

All the classes stored in the package `CBRSupport` are responsible for providing the infrastructure that is necessary to realize the main part of the functionality of the CBR cycle [2]. `CBRSupport` contains classes for case representation, case base handling, case-based retrieval and adaption, and similarity computation.

Case Representation

Class `Case` is the base class to represent any kind of case-based concepts. In its current implementation, however, it does not provide any functionality. `Case_HetPlayerApplication` is derived from class `Case` and embodies the case structure of cases in the domain of applying heterogeneous player types. That means, it is implemented in accordance with the domain model introduced in Section 3.1 and it depicts the C++ realization of the case structure shown in Figure 3. Thus, it includes a problem part in form of an array of available player types[12] and instance variables telling how many players have been used of which player type.

The solution part, on the other hand, is implemented as a vector of instances of class `SingleSolutionOfCase`, since several solutions may correspond to one

[12] Class `HeteroPlayerType` was already part of the existing implementation of the online coach.

problem part. An instance of `SingleSolutionOfCase` comprises those information in its instance variables that are essential to fulfill the specification given in Definition 4. In that definition a single solution s^C of a case C is defined as an 8-tuple $s^C = (form, mstrat, assign, opp, score, signif, stat, eval)$. We mapped the components of that tuple to an adequate C++ implementation as follows:

$form$	$\rightarrow$	instance of class `Formation` which is part of the existing implementation of the online coach
$mstrat$	$\rightarrow$	instance of class `MainStrategy` (see below)
$assign$	$\rightarrow$	array telling which player type was used for which player
opp	$\rightarrow$	name of the opponent team as a string
$score$	$\rightarrow$	two unsigned integer values
$signif$	$\rightarrow$	a float value; the length of the respective match is used as significance value
$stat$	$\rightarrow$	a record of float values for each team, representing the team statistics
$eval$	$\rightarrow$	a float value which is computed by the team performance evaluation function (according to Definition 8)

The class `MainStrategy` constitutes the representation of a main strategy, like for instance "4-3-3". To be accurate, the numbers of goal keepers and sweepers are included, too, so that an instance of `MainStrategy` encapsulates five unsigned integers. One important point concerning this class is that it contains a constructor that accepts an instance of class `Formation` as a parameter and that is able to infer the main strategy from the formation the considered team used.

Case Base Handling

An instance of class `CaseBase` is a container holding a set of cases. Its elements do not necessarily have to be instances of class `Case_HetPlayerApplication`. Indeed, a `CaseBase` may contain cases of any kind, as long as they are derived from the case concept class `Case`.

To get access to a specific case base, however, one needs a `CaseBaseInterface`. Instances of that class are able to manage the access to case bases of different kinds. Such an interface to a case base provides functionality such as opening and closing case bases as well as adding, removing, or listing cases.

Since the case representation is object-oriented and attribute-value based, the XML format [10] turned out to be very appropriate for writing out cases to file and reading them in, respectively. Using that uniform standard format instead of writing out cases in a proprietary binary format, the contents of a case base may be used easily by other programs and for other purposes, too. In Appendix A we give the XML Document Type Definition (DTD) for the structure of a case.

Retrieval and Adaption

An instance of class `CBRRetrievalUnit` mainly provides a set of retrieval methods expecting different parameters to direct the retrieval process as desired by the user. Internally, this class, at its current stage of development, realizes any kind of retrieval as *linear retrieval*. That means, using an instance of `CaseBaseInterface`, the retrieval unit iterates linearly over all cases in the particular case base and determines for each case the similarity to the query. Having finished the retrieval, it returns an instance of class `RetrievalResult`. An object of that container class summarizes the result of a retrieval, which of course fundamentally depends on the parameters used during the retrieval (see C for more details).

A `CBRAdaptionUnit` is able to adapt an existing solution (pertaining to an old case) so that it can be applied to the current situation. To do so it implements the substitutional operator pt^* introduced in Definition 11. Accordingly, instances of that class are capable of generating suggestions saying which players should be substituted for which player types.

Similarity Measures

All similarity measures we need to guarantee the correct operation of the whole CBR framework are arranged in a hierarchical inheritance class structure. That way it is easier to change, extend, and maintain existing or to add new similarity measures. `SimilarityMeasure` represents the abstract base class to any kind of similarity measure. It provides an abstract method `getSimToQuery(case,query)` whose functionality has to be (re-)implemented in each subclass of `SimilarityMeasure`, depending on the semantics of the respective similarity metric.

Class `WeightedSimMeasure` depicts a similarity measure whose attribute similarities are amalgamated after being weighted. From that class we inherit the `WeightedSimMeasureBetweenPlayerTypes` which stands for a metric that specifically handles the computation of the similarity between heterogeneous player types. Thus, it realizes the similarity measure sim_{HPT} as defined in Definition 10 and as optimized in Section 5.2. An instance of `WeightedSimMeasureBetweenPlayerTypes` reads the player attribute weights from a weight file that has to be specified as a coach parameter (see Appendix C).

Another subclass of `SimilarityMeasure` is `UserDefinedSimMeasure`. That class represents the base class for the `SimMeasureBetweenPlayerTypeSets`. The latter one realizes the computation of the similarity between queries and top-level cases in the domain of heterogeneous player types: the case similarity Sim. The semantics of that measure is given in Definition 12.

A third subclass of `SimilarityMeasure` is the `FloatSimMeasure`. That simple metric computes the similarity between two real values within a specified interval. It serves to calculate the attribute similarities of a heterogeneous player type's characteristics as given in Definition 9 (player attribute similarity sim_A).

6.3 Module ModCaseBaseManagement

This module provides basic functionality to create new cases, to add them to an existing case base, or to create a new case base. `ModCaseBaseManagement` analyzes an existing logfile of a simulated soccer match and extracts all the necessary information to create a summarizing case (an instance of class `Case_HetPlayerApplication`) that contains all the relevant information on the match and on the use of heterogeneous player types.

The considered module as part of the online coach expects an existing logfile as input and a case base as well. If there is no existing case base specified, it will create a new one. Furthermore, `ModCaseBaseManagement` is derived from class `StatWatcher` (statistic watcher). In cooperation with the already implemented coach module `ModProxyServer` it makes use of the Statistics Proxy Server to obtain statistical information on the match. `ModProxyServer` fetches the current statistics from the Statistics Proxy Server and informs the module `ModCaseBaseManagement` about changes in their values via the functionality provided by class `StatWatcher`.

After having parsed the logfile, `ModCaseBaseManagement` integrates the team statistics on each of the two teams, that participated in the match, into the case to be constructed. Besides, it utilizes them to assess the respective team performance, using the team performance evaluation function that we introduced in Definition 8.

6.4 Module ModCBRHeterogeneousPlayers

This module is used to utilize heterogeneous player types for an upcoming match in a CBR-based manner. When the game is started the specified case base will be scanned for a case that is similar to the current situation, i.e. to the currently available, predetermined player types.

The way of retrieving a similar case and deciding for an old solution with an instance of `CBRRetrievalUnit` can be influenced by several parameters that are described in more detail in Appendix C. If the retrieval returns a retrieval result with more than one entry, `ModCBRHeterogeneousPlayers` decides for that solution that features the maximal product of its evaluation value and its similarity to the query, i.e. to the current situation.

After the module at hand has decided for an appropriate old solution, it is adapted by a `CBRAdaptionUnit` so that it can be employed for the match at issue. That means, the `ModCBRHeterogeneousPlayers` acquires a suggestion saying which player types should be used for which players and conducts the corresponding player type substitutions.

7 Conclusion and Future Work

Participating teams in simulated robotic soccer matches are permitted to employ varying player types, i.e. types that differ from each other in their physical attributes. The decision which of these predetermined heterogeneous player types should be used best for the team line-up in an upcoming match behooves the respective team's online coach.

We have introduced an approach that enables the coach to solve that problem by using techniques that base on case-based reasoning. Applying CBR in this domain facilitates the reuse of experiences made during former soccer games. Moreover, by considering team line-ups as a whole, we can indirectly also regard synergy effects that come into existence by applying certain combinations of player types.

We have enhanced the online coach of Carnegie Mellon's simulated robotic soccer team ChaMeleons-01, so that is is capable to realize the whole CBR cycle (retrieve–adapt–reuse–retain [2]). Thus, it is able to substitute and apply heterogeneous player types in a CBR-based way.

An integral part of this work deals with the assessment of case solutions. A meaningful evaluation of the team performance, that is yielded with the help of applying particular player types, is crucial for the reutilization of old cases. Using data mining techniques, we have analyzed logfiles of past matches and found out which factors influence the success or failure of a team, i.e. winning or losing, primarily. Here, it was our goal to acquire a differentiated team performance estimation, which is more sophisticated than a crude distinction like "win–draw–lose".

A further part of this report handles the automated learning of the similarity measure between heterogeneous player types. Applying a machine learning approach, we have focused on the question which characteristics of a player type have a significant impact on the similarity calculation between two player types, and which attributes are less relevant. In addition to that, this analysis partly revealed which player characteristics have an increased influence on the player's behaviour in a game and on its playing skills.

The CBR framework we implemented is entirely operative and ready to use.

The successful application of CBR in general presupposes that experience knowledge in form of cases in a comprehensive case base is available. However, the assembling of such a case base was not part of the scope of this work. Thus, that can be considered as one important future task.

On the one hand it is possible to build up an extensive case base by conducting a large number of simulated test matches using varying settings for the participating teams. Admittedly, that proceeding would require very much computing time. On the other hand, there is already a collection of logfiles resulting from the matches of the last RoboCup tournaments. However, the majority of the participants did not make use of heterogeneous player types at all, just relying on the default player type. As far as upcoming RoboCup events are considered, it may be expected that more and more teams are about to employ heterogeneous player types. Hence, the next tournaments might prove to be suppliers of case knowledge.

In that context, possible future extensions may become necessary in the range of case base maintenance, management of several case bases, and problems concerning an efficient retrieval. In these respects our CBR framework features only rudimentary capabilities.

At its current stage of development the online coach chooses certain heterogeneous player types only prior to the upcoming match. However, the coach is also permitted to substitute a particular number of players (usually 3) during the game, i.e. to change their player types. Consequently, another future coach enhancement could be to determine these substitutions with the help of the team line-ups in the case base, too, depending on the current score and the opponents behaviour, for example.

Finishing we want to remark, that the created infrastructure can be deployed for other purposes as well. The possibility to generate a summary on a soccer match in form of a case, to evaluate the outcome of a game, and to export the gathered knowledge to the XML format lay the foundations to also use the obtained data for ulterior statistical analysis, as input for machine learning algorithms in the context of robotic soccer, or to appraise the skills of a team.

A DTD of a Case

This section provides an XML Document Type Definition for the structure of a case. Case base files as created by the coach module `ModCaseBaseManagement` are XML files that comply with this DTD.

```
<!ELEMENT  CASE_HET_PLAYER_APPLICATION
           (
             AVAILABLE_PLAYER_TYPES,
             NUMBER_OF_USED_HET_PLAYER_TYPES,
             NUMBERS_OF_USED_PLAYERS_OF_TYPE,
             SOLUTIONS
           ) * >
<!ELEMENT  AVAILABLE_PLAYER_TYPES, (HETERO_PLAYER_TYPE) + >
<!ELEMENT  HETERO_PLAYER_TYPE
           (
             INDEX,
             PLAYER_SPEED_MAX, STAMINA_INC_MAX, PLAYER_DECAY,    INERTIA_MOMENT,
             DASH_POWER_RATE,  PLAYER_SIZE,     KICKABLE_MARGIN, KICK_RAND,
             EXTRA_STAMINA,    EFFORT_MIN,       EFFORT_MAX
           ) >
<!ELEMENT  INDEX               (#PCDATA) >
<!ELEMENT  PLAYER_SPEED_MAX    (#PCDATA) >
...
<!ELEMENT  EFFORT_MIN          (#PCDATA) >
<!ELEMENT  NUMBER_OF_USED_HET_PLAYER_TYPES  (#PCDATA) >
<!ELEMENT  NUMBERS_OF_USED_PLAYERS_OF_TYPE  NUM_OF_PLAYER_TYPE >
<!ELEMENT  NUM_OF_PLAYER_TYPE  (#PCDATA) >
<!ATTLIST  NUM_OF_PLAYER_TYPE
           index  CDATA [REQUIRED | 0] >
<!ELEMENT  SOLUTIONS (SOLUTION) + >
<!ELEMENT  SOLUTION
           (
             FORMATION,             MAINSTRATEGY,
             EVALUATION,            OUR_TEAM_STATISTICS,
             THEIR_TEAM_STATISTICS, USED_PLAYER_TYPES_FOR_PLAYERS,
             INFERRED_PLAYER_ROLES, OPPONENT,
             OUR_SCORE,             THEIR_SCORE
           ) >
<!ELEMENT  FORMATION           (#PCDATA) >
<!ELEMENT  MAINSTRATEGY
           (
             NUMBER_OF_GOALTENDERS, NUMBER_OF_SWEEPERS,  NUMBER_OFDEFENDERS,
             NUMBER_OF_MIDFIELDERS, NUMBER_OF_FORWARDS
           ) >
<!ELEMENT  NUMBER_OF_GOALTENDERS (#PCDATA) >
...
<!ELEMENT  NUMBER_OF_FORWARDERS  (#PCDATA) >
<!ELEMENT  OUR_TEAM_STATISTICS
           (
             TXAvr, TYAvr, TXVar,
             ...
             TOffSideNum
           ) >
<!ELEMENT  TXAvr               (#PCDATA) >
...
<!ELEMENT  TOffSideNum         (#PCDATA) >
```

```
<!ELEMENT  THEIR_TEAM_STATISTICS
           (
             TXAvr, TYAvr, TXVar,
             ...
             TOffSideNum
           ) >
<!ELEMENT  TXAvr                (#PCDATA) >
...
<!ELEMENT  TOffSideNum          (#PCDATA) >
<!ELEMENT  USED_PLAYER_TYPES_FOR_PLAYERS
                                (#PCDATA) >
<!ATTLIST  USED_PLAYER_TYPES_FOR_PLAYERS
           index  CDATA [REQUIRED | 0] >
<!ELEMENT  INFERRED_PLAYER_ROLES (#PCDATA) >
<!ATTLIST  INFERRED_PLAYER_ROLES
           index  CDATA [REQUIRED | 0] >
<!ELEMENT  OPPONENT             (#PCDATA) >
<!ELEMENT  SIGNIFICANCE         (#PCDATA) >
<!ELEMENT  OUR_SCORE            (#PCDATA) >
<!ELEMENT  THEIR_SCORE          (#PCDATA) >
```

B Description of Available Team Statistics

In total, the Statistics Proxy Server generates 33 team statistics on a match. In the following, we give a listing of these statistics together with a short explanation.

The preceding "T" in each statistic's name relates to the fact that these are *team* statistics, in contrast to *player* and *correlative* statistics that the proxy server can also generate.

- **TXAvr:** average X-location of all players
- **TYAvr:** average Y-location of all players
- **TXVar:** deviation of all players' X-location
- **TYVar:** deviation of all players' Y-location
- **TPosession:** ball possession rate of a team
- **TPassNum:** total number of passes
- **TPassLenAvr:** average length of passes
- **TPassLenVar:** deviation of pass lengths
- **TPassLongNum:** total number of "long" passes; the Statistics Proxy Server determines what kind of passes are to be consideres as "long" ones
- **TPassBackNum:** total number of backward passes
- **TPassSuccessRate:** average pass success rate
- **TShotNum:** total number of shots onto the opponent goal
- **TDribbleNum:** total number of dribbles
- **TStealNum:** total number of steals
- **TScore:** the team's score
- **TShootSuccessRate:** percentage of successful shots on the opponent goal
- **TDistanceAvr:** average distance covered by one player
- **TPassChainNum:** total number of pass chains
- **TPassChainPlayerNumAvr:** average number of players that are involved in a pass chain
- **TPassChainLenAvr:** average length of pass chains

- **TDribbleLenAvr:** average length of dribbles
- **TInactivePlayerNum:** number of players making no passes at all
- **TCompactnessAvr:** average X-distance between the team's front-most and its rear-most player (excluding the goalkeeper)
- **TCompactnessVar:** deviation of the team compactness
- **TBallPlayerDisAvr:** average distance of ball and players
- **TBallPlayerDisVar:** deviation of the distance between ball and players
- **TWinningPassPatternNum:** total number of winning pass patterns
- **TBallAtEachSide:** territorial advantage; time share during which the ball is in the opponent half of the field
- **TCornerKickNum:** total number of corner kicks
- **TGoalKickNum:** total number of goal kicks
- **TFreeKickNum:** total number of free kicks
- **TKickInNum:** total number of kick ins
- **TOffSideNum:** total number of offsides

C Introduced Coach Parameters

The coach's modules are controlled via a set of parameters that are specified in the coach configuration file `coach.conf`. For the two new coach modules we implemented we had to add several new parameters.

The following coach parameters are associated with the module `ModCaseBaseManagement`:

- **do_case_base_management:** off / on
  General parameter to switch the use of this module on or off.
- **use_case_base_management_menu:** off / off
  This menu with its basic functionality shall help maintaining cases within a case base.
- **case_base_fn:** string
  This parameter specifies the name of the case base in which to store the cases or from which to read cases. When the referenced file does not exist, a new case base will be created.
- **case_equality_threshold:** float $\in$ [0,1], (default value: 1.0)
  This parameter defines a threshold that determines when cases are considered to be equal. If the similarity of the case c_{new}, that is about to be added to a case base, to a case c_{old} in the case base exceeds this threshold, the information extracted from the logfile will be added as a new solution of c_{old} instead as being added as a whole new case.
- **side_to_analyze:** left / right
  This parameter decides which of the both teams that joined the game has to be analyzed. A new case will be created for that team.

The following coach parameters are associated with the module `ModCBRHeterogeneousPlayers`:

- **use_player_types_cbr_based:** off / on
  General parameter to switch the use of this module on or off.
- **case_base_fn:** off / on

This parameter specifies the name of the case base in which to search for old cases and solutions.

- **sim_measure_between_ptypes_weights_fn:** string
  This represents the name of the file in which the weights for the similarity measure between player types are specified. Since the similarity measure between heterogeneous player types is a weighted similarity measure, a weight for each property of a player type has to be specified (as a float value).
  In case that there is no weight file specified or that the weight file cannot be opened, default weights (uniform weight vector) will be used.
- **cbr_pt_case_min_sim:** float
  This parameter influences the retrieval behaviour of this module. When searching the case base for cases that are similar to the current situation, only those ones will be considered that have at least a similarity of cbr_pt_case_min_sim.
- **cbr_pt_solution_min_eval:** float
  This parameter influences the retrieval behaviour of this module. When searching the case base for cases that are similar to the current situation, only those solutions will be taken into consideration that have at least an evaluation value of cbr_pt_solution_min_eval.
- **cbr_pt_solution_min_significance:** float
  This parameter influences the retrieval behaviour of this module. When searching the case base for cases that are similar to the current situation, only those solutions will be taken into consideration that have at least a significance value of cbr_pt_solution_min_significance.
- **cbr_pt_retrieval_result_max_size:** float
  This parameter influences the retrieval behaviour of this module. When retrieving the case base for cases that are similar to the current situation, a retrieval result will be returned that maximally contains the most similar cbr_pt_retrieval_result_max_size cases.
- **cbr_pt_desired_main_strategy:** format string [13]
  This parameter influences the retrieval behaviour of this module. When retrieving the case base for cases that are similar to the current situation, only those old solutions will be taken into consideration whose corresponding match was lead by using a main strategy that matches cbr_pt_desired_main_strategy.
- **cbr_pt_opponent_team_name:** string
  This parameter influences the retrieval behaviour of this module. When retrieving the case base for cases that are similar to the current situation, only those old solutions will be taken into consideration whose corresponding match represented a game against a team named cbr_pt_opponent_team_name.

[13] The format string is a 5 digit number in which each digit stands for the number of players of a player role (first digit for number of goalies, second for number of sweepers, third for the number of defenders, fourth for the number of midfielders and the fifth digit for the number of forwarders. "10433", for example, represents a very typical main strategy.

References

[1] R.B. Bendel and A.A. Afifi. *Comparison of Stopping Rules in Forward Stepwise Regression.* Journal of the American Statistical Association 72, pp. 46-53, 1977.

[2] R. Bergmann. *Grundlagen fallbasierter Systeme.* Lecture Notes on University Lecture 89-186, University of Kaiserslautern, 2000. At URL: http://wwwagr.informatik.uni-kl.de/ bergmann

[3] P. Carpenter, P. Riley, G. Kaminka, M. Veloso, I. Thayer, and R. Wang. In A. Birk, S. Coradeschi, and S. Tadokoro, editors. *RoboCup-2001: The Fifth RoboCup Competitions and Conferences.* Springer Verlag, Berlin, 2002, forthcoming.

[4] M. Chen, E. Foroughi, F. Heintz, Z. Huang, S. Kapetanakis, K. Kostiadis, J. Kummeneje, I. Noda, O. Obst, P. Riley, T. Steffens, Y. Wang, and X. Yin. *Users Manual – RoboCup Soccer Server – for Soccer Server Version 7.07 and later.* RoboCup Federation, 2001, at URL http://www.ida.liu.se/ frehe/publications/manual-7.ps.gz

[5] M.C. Costanza and A.A. Afifi. *Comparison of Stopping Rules in Forward Stepwise Discriminant Analysis.* Journal of the American Statistical Association 74, pp. 777-785, 1979.

[6] R.A. Fisher. *Statistical Methods for Research Workers (6th ed.).* Oliver and Boyd, Edinburgh, 1936.

[7] I. Frank, K. Tanaka-Ishii, and K. Arai. *The Statistics Proxy Server.* Fourth International Workshop on RoboCup, Melbourne, Australia. Springer-Verlag, Lecture Notes in Computer Science - Artificial Intelligence series, 2000.

[8] I. Frank, K. Tanaka-Ishii, K. Arai, and H. Matsubara. *Statistics Proxy Server Manual v1.0.* At URL http://www.etl.go.jp/ ianf/Mike/ProxyManual.ps.gz, 2000.

[9] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*, pp. 1-88. Addison Wesley, New York, 1989.

[10] E.R. Harold. *XML Bible.* IDG Books Worldwide, Foster City, 1999.

[11] J.H. Holland. *Adaption in Natural and Artificial Systems.* The University of Michigan Press, 1975.

[12] D.W. Hosmer, and S. Lemeshow. *Applied Logistic Regression: 2nd Edition*, pp. 31-128. Wiley, New York, 2000.

[13] J. Kolodner. *Case-Based Reasoning.* Morgan-Kaufmann Publishers, 1993.

[14] D. Leake. *Case-Based Reasoning, Experiences, Lessons & Future Directions.* American Association for Artificial Intelligence Press, 1996.

[15] K.I. Lee and J.J. Koval. *Determination of the Best Significance Level in Forward Stepwise Regression.* Communications in Statistics: Simulation and Communication 26(2), pp. 559-575, 1997.

[16] T. Lim. *User's Guide for logdiscr Version 2.0.* At URL http://recursive-partitioning.com/logdiscr/guide.pdf, 1999.

[17] C.K. Riesbeck, and R.C. Schank. *Inside CaseBased Reasoning.* Lawrence Erlbaum Associates, Cambridge, 1989.

[18] H.P. Schwefel. *Numerical Optimization of Computer Models.* Wiley, Chichester, 1981.

[19] H.P. Schwefel, and T. Bck. *Artificial Evolution: How and Why?* In D. Quagliarella, C. Pariaux, C. Poloni, and G. Winter (ed): *Genetic Algorithms and Evolutionary Strategies in Engineering and Computer Science*, pp. 1-19. Wiley, Chichester, 1998.

[20] A. Stahl. *Learning Feature Weights from Case Order Feedback.* Proceedings of the 4th International Conference on Case-Based Reasoning (ICCBR), Vancouver, 2001.

[21] StatSoft Incorporated. *Electronic Statistics Textbook.* At URL http://www.statsoftinc.com/textbook/stathome.html, 2001.