# Improving the Performance of a RoboCup Case-Based Imitation Agent through Preprocessing of the Case Base

by

**Michael W. Floyd**

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
**Master of Applied Science**

Ottawa-Carleton Institute of Electrical and Computer Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
December 2008

The undersigned recommend to

the Faculty of Graduate Studies and Research

acceptance of the thesis

# Improving the Performance of a RoboCup Case-Based Imitation Agent through Preprocessing of the Case Base

Submitted by **Michael W. Floyd**

in partial fulfilment of the requirements for the degree of

**Master of Applied Science (Electrical Engineering)**

---

Thesis Supervisor
Dr. Babak Esfandiari

---

Chair, Department of Systems and Computer Engineering
Dr. Victor Aitken

Carleton University

2008

# Abstract

The transfer of knowledge from an expert to a software agent can be a tedious task. Instead, the agent can learn by observing the expert. This is done using case-based reasoning, with cases being composed of the expert's sensory inputs and performed actions. The agent can then imitate the expert by performing similar actions when presented with similar inputs. Case data is collected in an automated manner and large amounts of data can be collected inexpensively. However, if the imitating agent must operate in a real-time environment there is a limit to the number of cases they can realistically use.

This thesis examines, with RoboCup soccer data, how feature reduction and prototyping can be used to both increase the number of cases that can be searched and remove redundant cases. Experimental results have shown preprocessing the case base can significantly improve the imitative ability of an agent. For each method of preprocessing, various algorithms are compared and the algorithms most beneficial for RoboCup data are identified.

# Acknowledgments

I would like to thank Professor Babak Esfandiari for his support and guidance through every stage of this process. In addition to his supervision on this thesis, he also provided excellent insight and feedback during the writing of several related publications.

I would also like to thank the other graduate students in my lab, Francois Gagnon, Alan Davoust and Edgar Acosta , for providing excellent feedback and discussions of ideas. Also, I would like to thank Ibukunoluwa Ajila, Rohit Rishi, Hamish Robertson and Supun Wijenayake for helping with various programming, testing and data collection tasks.

Lastly, I would like to thank my family and Natalie. They put up with my busy schedule and odd hours while always encouraging me to do what I enjoy. Their understanding has made the entire process much easier and more enjoyable.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Designing and implementing software agents is often a difficult task that requires a significant amount of development time and software programming expertise [32]. In addition to the technical skills required to develop a software agent there is also the need to transfer domain knowledge to the agent, often times from a human expert. This requires modelling the expert knowledge in a manner that is interpretable by a software agent. Even after a software agent has been created, it is likely only able to perform a specific set of tasks in a given domain. Adding the ability to perform new tasks or deploying the agent in novel domains may require providing the agent with additional or modified expert knowledge.

In order to remove the technical skills required to model expert knowledge and train a software agent, and thereby making the software agent useful to people without such technical skills, the ability for an agent to learn through observation could be employed. More specifically, the agent could observe an expert (a human or another agent) perform a task and then attempt to imitate the behaviour of the expert when faced with a similar task. This type of approach moves the burden of modelling the expert knowledge from the expert to the agent.

## 1.1 Motivations

After observing how a teacher behaves in specific situations, the agent can attempt to behave similarly when presented with similar situations. The imitating agent may not have any knowledge of the reasoning process used by the teacher and may only be able to observe the action the teacher performs, $a$, in response to sensory inputs, $x_1 \ldots x_n$. The teacher could then be modelled by a function as follows:

$$a = f(x_1, \ldots, x_n) \tag{1.1}$$

By remembering the actions the expert performed given the sensory inputs, the agent could then perform similar actions when encountering similar sensory inputs. This type of reasoning, by solving current problems (sensory inputs) using knowledge from previous problem instances, is known as case-based reasoning [1]. Case-based reasoning exploits the assumption that similar problems have similar solutions, so it lends itself well to imitation where an agent and teacher should behave similarly given similar sensory inputs.

One limitation of this type of approach is that the reasoning ability of the system is related to the number of past experiences, called *cases*, that can be remembered. The case-based reasoning system must contain enough cases so that a wide variety of input problems can be solved. As the number of stored cases increases, so does the computational effort required to search for cases similar to the current input problem. This problem is especially prevalent in real-time domains where the agent has a fixed amount of time to perform case-based reasoning.

Using case-based reasoning to imitate the behaviour of software agents has previously been applied [31] in the domain of simulated RoboCup soccer (more details on RoboCup in Chapter 2). In RoboCup, each soccer playing agent receives sensory information about their environment and can in turn perform actions like dashing,

turning and kicking. By observing an agent playing RoboCup soccer, a collection of cases, with each case containing the agent's sensory inputs and performed action, can be generated in an automated manner. During run-time, an imitative agent compares its current sensory inputs to the sensory inputs contained in the cases and reuses the actions from similar cases in attempt to mimic the original agent.

While such a case creation approach allows a large collection of cases to be easily generated it also causes complications since an imitative agent can only use a limited number of cases due to real-time constraints. As the number of cases used by the imitative agent increases so to does the amount of time it takes to search for similar cases. If the search time becomes too large, it may cause the imitative agent to react slowly to changes in the environment.

### 1.1.1 Real-time Concerns

Software agents are, in many situations, required to operate in environments where they must act in real time. This is certainly true for simulated robotic soccer players who have the ability to perform a single action per time interval, called a *cycle* [12] in RoboCup soccer. Each cycle they receive sensory information from a server, which maintains the world model and acts as the referee for the soccer game, and can in turn send to the server the action they wish to perform. If they do not send their desired action to the server within that time interval they will miss out on an opportunity to perform an action, and may not perform as effectively as desired.

Such a real-time constraint is not only important for a simulated robotic soccer player, but it is also important for an agent trying to imitate the behaviour of the soccer player (or trying to imitate any other real-time agent). The imitative agent must compare its current sensory inputs (the state of the environment) to the stored cases and find the most similar case to the sensory inputs. The agent can then reuse the action associated with the most similar case. The process used to determine which

case is most similar, in case-based reasoning, often involves using a nearest-neighbour search or a variant of nearest-neighbour search [1]. Nearest-neighbour searches are known to have a high computationally load since the input query, the sensory inputs of the agent, must be compared to all[1] possible neighbours [25], the stored cases. As the number of neighbours increase, so does the time it takes to complete the nearest-neighbour search.

As the number of cases used by an imitative agent increases, it is likely that an associated rise in the agent's ability to imitate will occur. This creates a situation where a large case base is desirable in order to maximize imitative performance, yet the case base size is bound by the number of cases that can be searched within the real-time limits. This is demonstrated in Figure 1.1. As the number of cases in the case base increases, so to does the accuracy of the imitation system[2]. The dashed line shows the maximum case base size that can be used by the system given its real-time constraints (more on that in Chapter 5). It would therefore be desirable to increase the size of case base that can be searched within the real-time limit. However, simply using a larger case base does not necessarily increase the diversity of the case base. If a newly added case is highly similar to an existing case then the addition on the new case would likely not improve the imitative ability of the agent. The ability to identify highly similar cases and ensure that the case base does not contain sets of highly similar cases then becomes important. **A solution to this problem is to transform, or preprocess, the cases in the case-base in such a way that more cases or a more diverse case base can be searched within a real-time limit**.

[1]There are variants of traditional nearest-neighbour search that use case indexing or store the cases in a hierarchal structure. Although these variants do not require comparing the input to every case, the computational or storage requirements will still increase as the number of cases are increased.

[2]This data is for the Krislet agent using parameters that will be described in Chapter 4.

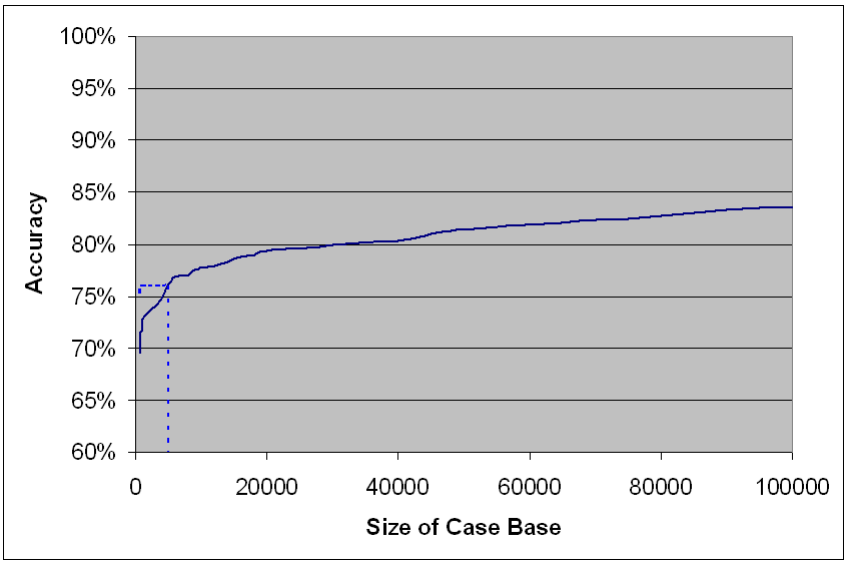


**Figure 1.1:** The imitation accuracy as the size of the case base is increased.

## 1.2 Objectives

The goal of this research is to preprocess a case base, used by an agent attempting to imitate a RoboCup soccer player, in order to reduce the amount of time it takes to search the case base without negatively impacting the diversity of data contained in the case base. This leads to the following research question:

**Research Question**: In the domain of simulated RoboCup soccer, how can the imitative ability of a real-time case-based imitation agent improve by preprocessing the case base it uses?

This research question addresses the general notion of preprocessing a case base used by an imitative agent. Numerous methods could potentially be used to examine the feasibility of preprocessing, however this thesis will limit itself to several methods. It order to address the main research question, the following questions will be examined:

1. **Feature Removal:** Will removing less relevant features from cases result in a

decrease in execution time, and by how much, without negatively affecting the imitative performance of a RoboCup imitation agent?

2. **Case Prototyping:** Can sets of similar cases, grouped using data clustering, be replaced with a single prototypical case without a significant negative effect on the performance of a RoboCup imitation agent? What level of compression can be achieved before negative performance occurs?

3. Does combining feature removal (1) and prototyping (2) result in improved performance compared to using each technique separately, and does the order of preprocessing matter?

## 1.3 Contributions

When generating cases in an automated manner by observing an expert, the cost of collecting data is quite low so it is possible to generate a large amount of data. However, in case-based reasoning as the number of cases in the case base increase so to does the number of instances that must be searched when attempting to solve a problem. The primary contributions of this thesis relate to preprocessing of large case bases, used by agents attempting to imitate RoboCup soccer players, in order to maximize the diversity of information contained in these case bases while meeting imposed time limits. The following lists several key contributions:

- Evaluation of various feature selection, clustering and prototyping method on data used by an agent imitating the behaviour of a RoboCup player.

- Conclusions that, in the domain tested, the following algorithms provided the best results:

  - *Feature Selection:* A binary feature selection algorithm with a dynamic sized training set (Chapter 5).
  - *Clustering:* Transforming the data using the distance vector approach and then using k-means clustering (Chapter 6).
  - *Prototyping:* Creating an average case (Chapter 7).
  - *Hybrid:* Performing feature reduction before prototyping (Chapter 8).

- A method of feature selection, that can be used with existing algorithms, that considers the cost of retaining features when selecting an optimal feature set.
- Development of variants of existing prototyping methods that handle the data used by a RoboCup imitation agent.
- An open-source software framework for agent imitation using case-based reasoning. This framework provides a domain-independent implementation so that it can be deployed in a variety of domains and is not restricted to RoboCup soccer.

The case base feature removal, clustering and prototyping techniques are shown, experimentally, to provide a significant improvement over an unprocessed case base. While these techniques do provide improvements, they still suffer similar shortcomings to previous case-based imitation work. The imitation framework is not able to identify multiple states of behaviour and only considers visual stimuli. Therefore the imitative agents have difficulty imitating multi-state agents or agents that rely heavily on inter-agent communication.

## 1.4 Organization

Following the introduction, Chapter 2 provides a brief overview of case-based reasoning and describes the RoboCup domain that will be used as a case study. Chapter 3 provides an overview of the current state of research in imitative learning, real-time case-based reasoning, and maintaining case bases. Chapter 4 outlines the methodological approach followed in this thesis, as well as the proposed unit of analysis, method of data collection and method of data analysis.

Removing features to decrease search time in case-based reasoning is presented in Chapter 5. Chapter 6 examines how the case base can be clustered and Chapter 7 discusses using these clusters to create prototypical cases. Feature removal and case prototyping techniques are combined in Chapter 8. Chapter 9 describes the software framework for agent imitation used in this thesis. Finally, Chapter 10 provides conclusions and outlines future areas of work.

# Chapter 2

# Background

The following sections will provide a brief summary of two important topics that are relevant to this thesis: the RoboCup Simulation League and case-based reasoning.

## 2.1 RoboCup Simulation League

RoboCup [43] is a collection of competitions that aim to foster artificial intelligence and robotics research. RoboCup initially started as a robotic soccer competition but has since expanded to include disaster recovery (RoboCup Rescue) and human-machine interaction (RoboCup@Home) competitions. RoboCup has become an increasingly popular platform for the application and testing of artificial intelligence, multi-agent systems and robotics research and is the focus of numerous conferences and workshops every year. Various algorithms and techniques can be compared using one of the RoboCup competitions as a common platform.

The RoboCup soccer competition itself is further divided based on the type of robots used. There are divisions for small-size, medium-size, four-legged and humanoid robots as well as a simulation division. The simulation division, called the RoboCup Simulation League, provides an excellent environment for research into artificial intelligence, machine learning and multi-agent systems without the added

| | **Parameters** |
|---|---|
| **Ball** | distance, direction |
| **Goal** | distance, direction, side |
| **Line** | distance, direction, identifier |
| **Flag** | distance, direction, identifier |
| **Player** | distance, direction, team, uniform number |

**Table 2.1:** RoboCup soccer simulation visible objects and their parameters.

overhead of building and maintaining physical robots.

The RoboCup Simulation League is based on a client-server architecture with each player represented as a separate client. All players connect to the soccer server [12] which maintains the state of the game (score, time, and location of objects), enforces the rules of the game and handles communication with each player. At discrete time intervals the server sends a message to each player informing them of their current sensory inputs (Table 2.1) and the players respond with the action they wish to perform (Table 2.2). A graphical representation of a simulated game can be seen in Figure 2.1.

RoboCup Simulation League provides a dynamic, non-deterministic environment where players have an incomplete world model. Other players (and the ball) are able to move and can influence the environment, so there is no guarantee a player's desired action will succeed (for example, a player may try to kick the ball but another player may have already kicked it). As well, players only have a limited view of the field (Figure 2.2) and they cannot be certain of the position of objects (due to noise in the position data provided by the server). These factors create interesting challenges that must be taken into account when designing how the players will reason and behave.

A variety of different approaches have been used to create RoboCup Simulation League players ranging from simple players that run toward the ball and kick it

| | Parameters | Notes |
|---|---|---|
| **Kick** | power (-100 to 100), direction (-90 to 90) | once per cycle |
| **Dash** | power (-100 to 100) | once per cycle |
| **Turn** | moment (-180 to 180) | once per cycle |
| **Catch** | direction (-90 to 90) | once per cycle, only by goalie |
| **Turn Neck** | angle (-90 to 90) | can be performed in the same cycle as other actions |
| **Move** | X (-54 to 54) and Y (-32 to 32) position | can only be performed before kickoffs |

**Table 2.2:** RoboCup soccer simulation actions and their parameters.

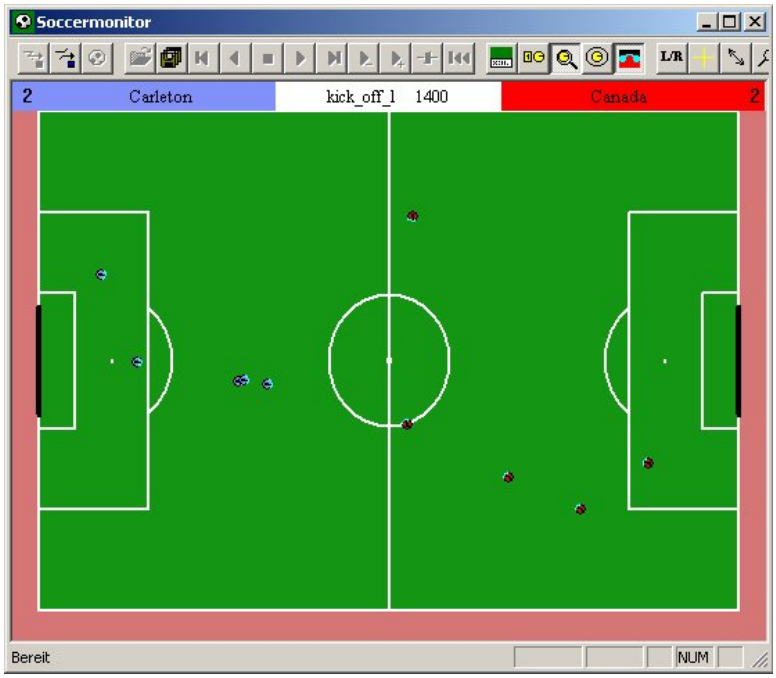


**Figure 2.1:** A visualization of a RoboCup Simulation League game.

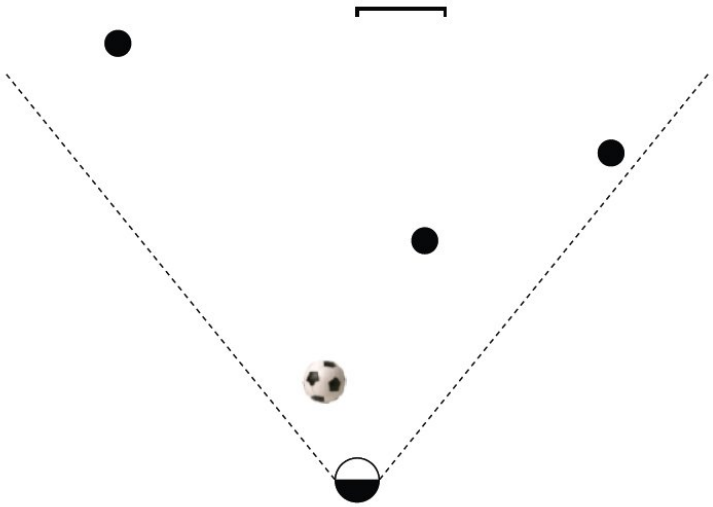

**Figure 2.2:** The field of vision of a RoboCup Simulation League player.

at the goal [33] to more complex teams that use state machines [17], reinforcement learning [42, 49] and layered learning [53] techniques.

## 2.2 Case-Based Reasoning

Case-based reasoning (CBR) makes use of the idea that similar problems have similar solutions [1]. CBR uses a collection of past problems with known solutions in order to determine what the solution to a novel problem is. This form of reasoning is comparable to when a human problem solver recognizes a set of criteria that are similar to a problem they have encountered previously. They may be reminded of the solution they used to solve that prior problem, and apply it to the current situation.

We define the following terms:

**Case**: A previously encountered problem and the associated solution to that problem.

**Case base**: A collection of cases.

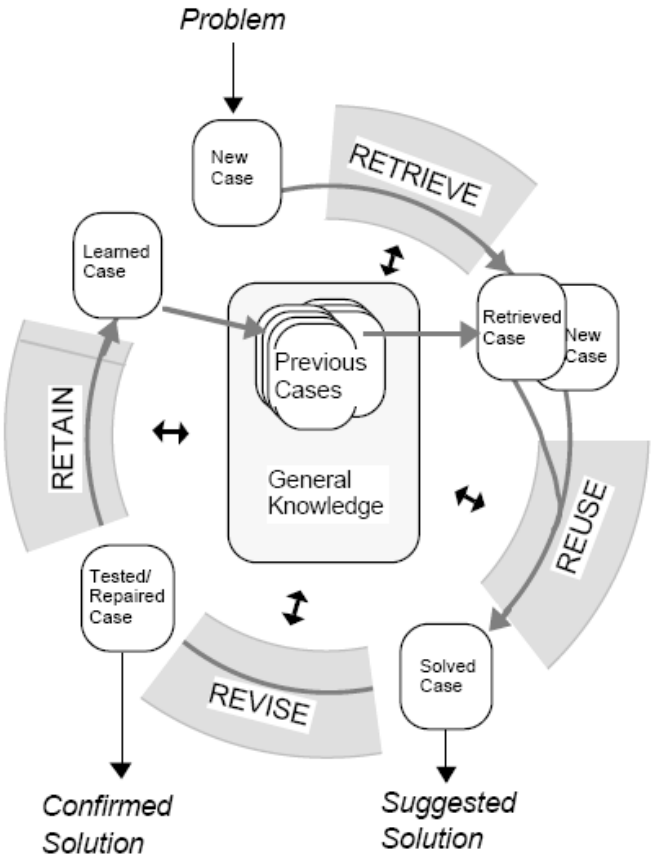


**Figure 2.3:** The case-based reasoning cycle. (Source: Aamodt and Plaza, 1994 [1]).

The case-based reasoning cycle begins when the CBR system is presented with a problem it must solve and the end results is a solution that can be applied to that problem. The CBR cycle involves four major steps [1] (Figure 2.3):

1. **Retrieve**: The retrieval stage involves finding the cases in the case base that are most similar to the current problem. This involves using a similarity function (which will depend on the domain and the nature of the way a problem is encoded) to compute the similarity between the current problem and all cases in the case base. The cases (the number of cases can vary depending on the usage, and could even just be a single case) deemed to be most similar can then be used by the Reuse stage.

2. **Reuse**: After the most similar cases have been retrieved it is then necessary to select the solution that will be used for the current problem. The solution used can be identical to a solution from one of the similar cases, a modified version of the solution from one (or more) of the similar cases or a new solution (if none of the similar solutions were appropriate). The solution is then applied to the problem.

3. **Revise**: In some domains it is possible to immediately see if an executed solution resulted in the desired results. If the solution solved the problem then no further changes are needed, however if the solution did not solve the problem then it will need to be revised. This revision could be a slight variation of the previously used solution or a complete change. The revised solution can then be applied (unless the environment has changed) and the success of the revised solution can be assessed (possibly leading to further revision).

4. **Retain**: If the applied solution (or a revised solution) successfully solved the problem, then it is possible to retain the problem-solution pair as a new case and add that case to the case base. The original case base was essentially a repository of past experiences (problems and solutions), so it makes sense to add new experiences and learn from them. This stage allows the case base to grow and cover a wider subspace of possible problems. One obvious issue with this stage is that as more cases are retained, the computational requirements during the retrieval stage will increase. This problem is addressed in a number of ways including setting a maximum case base size, only adding cases that are significantly different from the current cases or by not retaining any new cases at all.

# Chapter 3

# State of the Art

This chapter with explore several key areas that are related to the work performed in this thesis. Initially, various techniques for imitating humans, robots and software agents will be presented. Following that, the application of case-based reasoning in real-time domains, with specific focus on imitation, will be discussed. Lastly, the topic of case base maintenance will be examined in order to identify how a case base can be processed so that it can be used in a real-time domain.

## 3.1 Imitation

A number of approaches for imitation have appeared in the literature in various domains. In this section, a summary of such techniques will be discussed with a specific focus being placed on the imitation of software agents, robots and humans. This work is generally referred to as *programming by demonstration* or *programming by example* and has shown some interesting results involving the imitation of specific actions.

Imitation has been used to teach a robotic arm the pendulum swing-up task [6], which involves swinging a hanging pendulum upward and balancing the pendulum upright above the robotic arm, using a Kalman filter. This task is interesting because

it is actually composed of two separate tasks: swinging the pendulum to an upright position and balancing the pendulum. A human demonstrates this task, only moving along the horizontal plane, and the imitator uses a camera to log the movement of the human arm and pendulum.

Unlike many previous works on robot imitation [8], the demonstrated data could not simply be reused or the imitation would be unsuccessful due to the differences between a human and robotic arm. Instead, the imitator would need to learn a model for the task, using this demonstrated data, that was valid for its dynamics rather than the dynamics of the demonstrator. In their experimentation, they found that although using these models resulted in better performance than direct imitation there were still many instances where the swing-up task could not be successfully completed, likely due to model errors and noise in the demonstrated data. Although the imitative approach presented had limitations, it has shown itself to be valuable with many future works building on these foundations.

One such work that builds on the swing-up imitation involves imitating aerobatic manoeuvres performed by a robotic helicopter [13]. This work uses a variation of a Kalman smoother to perform the learning task. One of the primary contributions of this work is that it learns from multiple demonstrations. Also, it often performs the demonstrated tasks better then the original demonstrators. This is attributed to the fact that a human controller may perform a specific task with slight imperfections during a single demonstration but these imperfections are unlikely to occur in all demonstrations. The result of this is that the imperfections are eliminated when learning from all of these demonstrations.

For each demonstration, the trajectory of the helicopter is logged while it is controlled to perform aerobatic manoeuvres by a human. The length of time a manoeuvre takes can vary between demonstrations, so initially the various demonstrations must

be aligned so that similar parts of the demonstrated manoeuvres are combined together. Learning these alignments involves an iterative approach involving two steps. First, the alignments are set to constant values and the alignment parameters and ideal trajectories are computed. Secondly, the alignment parameters and ideal trajectories are set to constant values, from the previous step, and the alignments are computed. This process is repeated for a fixed number of iterations and then the aligned demonstrations are combined using weighted regression to create the final trajectories to be used.

Their experiments show successfully imitation of manoeuvres, such as flips or rolls, and also a series of manoeuvres. However, they also include expert knowledge in their learning process like that the central position of the helicopter should not move during flips. While this added expert knowledge allows for accurate imitation, the imitation would not work if this expert knowledge was not known or if it was incorrectly entered. For example, a designer may not know that the helicopter's central position should not move during a flip or the demonstrator may wish to perform a flip while the helicopter's centre is actually moving. Also, while a series of manoeuvres can be imitated they can only be imitated in the demonstrated order.

In a similar domain as the case-based imitation work, imitation has also been used on soccer playing dog robots [22]. This work mixes the ideas of programming by demonstration with *mixed initiative control* (MIC). MIC is where a robot can have aspects of its behaviour controlled by both an autonomous system and by a human, possibly at the same time. In the soccer robot application, the teacher and imitator are both able to control the robot. If the teacher actively controls the robot its commands are used and the imitator learns from the teacher's commands. However, if the teacher is not actively controlling the robot then the commands selected by the imitator are used.

The initial experiments train the imitator using a Locally Weighted Projection

Regression algorithm and show success when imitating the constant movement of the robots head and tail (moving them from side to side) and following the ball around. A hard-coded controller is used as the teacher, and a human controls when the teacher is active. After an initial period of learning, several additional shorter learning sessions are used when the imitator begins having troubles imitating. Additional experiments [23], using a Spare Online Gaussian Processes algorithm, attempt to imitate the actions of trapping the ball under the robot's head as well as walking toward the ball and then trapping it. In both studies the imitators had trouble with combined actions like walking toward the ball and then trapping it. In order to complete this task, they added an artificial input to the imitator related to the current task the teacher is performing (either walking to the ball or trapping the ball).

Imitation has also been used to control an agent in a *first-person shooter* video game [56]. Game logs, that contain the stimuli received by the agent and the actions performed by the agent, are gathered automatically by observing a hard-coded agent play the video game. Each stimulus is paired with the action performed after it to create a set of input-output pairs, and those pairs are then used to train a neural network. This neural network, which is trained offline, can then be used by the imitating agent while playing the video game. The approach only uses an expertly selected set of features to represent the input stimulus (the agent's position, distance to nearest opponent, vertical angle to nearest opponent and horizontal angle to nearest opponent) and also only use a subset of possible agent actions (related to moving and aiming the agent). They found success in imitating the tasks of moving toward an opponent and aiming in the direction of an opponent, however when watching the imitating agent in action there were often problems with the aiming task even using expertly selected features.

### 3.1.1 Lessons Learned

When examining the literature related to imitating agents, robots and humans, the following remarks can be made:

- In all of the discussed works, the imitators achieve their best success when only attempting to imitate a single action or task. Their training algorithms can be used to imitate a number of different tasks but only when those tasks are clearly separated. This can be seen in the difficulty of the robot arm performing the swing-up task and in the soccer playing robot moving toward and trapping a ball. For these tasks, the various subtasks can be imitated successfully by themselves however when they are attempted in series problems arise. With the robotic helicopter, a series of manoeuvres can be performed in series however it does not learn the individual manoeuvres but the series as a whole. The imitator would not be able to change the order of manoeuvres.

- The various techniques involve using some amount of expert knowledge. In the case of the robot helicopter, this involves hard-coding constraints related to various manoeuvres like flips. With the robotic arm a known model of a the dynamics of a robotic arm is used and with both the robotic dog and first-person shooter agent the inputs used for training are expertly selected. In addition, the robotic dog actually introduces an artificial feature to the environment in order to assist in performing a series of actions. This adds a cost to applying these techniques to new tasks or novel domains.

## 3.2 Case-based Reasoning in Real-time Domains

The imitative techniques described in the previous section have several key limitations, namely the need for expertly defined features and the inability to imitate

learned actions in an arbitrary order. In order to attempt to overcome these limitations we will examine another learning approach, case-based reasoning. Case-based reasoning has been applied in a number of real-time domains and, more specifically, has been used to imitate the behaviour of an expert. This work will be examined to not only identify how CBR is used for imitation but to also see how the real-time constraints of these systems are met.

### 3.2.1 Case-based Reasoning in Games

In a real-time military strategy game, case-based reasoning has been used to control the movement of units on a simulated battlefield by combining CBR with reinforcement learning [38]. This work makes use of two case bases. The first, the transitional case base, uses the current state of the environment to determine an action to perform and the new state that will arise from performing that action. The second, the value case base, uses a state to determine the expected value of entering that state. Various candidate solutions from the transitional case base can then be compared to see which one will produce a state with the highest value.

Each case base is initially empty, and grows during game play. If the transitional case base is empty or returns an incorrect state estimate, a new case is added. For the value case base, if the selected case was not similar enough to the input case then a new case is created. However, if the case was significantly similar to the input, the value attribute is updated using reinforcement learning. Since the case base is reset to empty at the beginning of each run, the case base never grows to a size that would require a large search time. This removes the need to perform any preprocessing on the case base but also results in a loss of everything that has been learned.

Reinforcement learning has also been combined with case-based reasoning in the domain of the game Tetris in order to learn game behaviour from a software agent [44]. Tetris involves stacking tiles of various shapes in order to create horizontal lines, from

one side of the game region to the other, with no gaps. Each case is composed of several input items: the current tile, the orientation of that tile and the orientation of the game region. The case solution contains where the tile should be moved, how it should be rotated, the value of the case and a usage metric for the case. The value of a case is used to rank different candidate case in order to select the best solution, whereas the usage metric is used to identify cases that are not often used so they can be removed if the size gets too large.

In a first-person shooter game, case-based reasoning has been used to control the movement of a team of players to specific locations on a map [7]. The goal of the game is to control specific locations, in this paper three different locations, on the map and keep the opponents off of those regions. In this approach, the players are controlled using a Q-table. The rows of the Q-table represent who controls each of the locations (team, opponent or unoccupied) and the columns represent which location each of three teammates should move to. The values in each matrix cell represent the expected score related to moving the teammates to those locations when the locations are currently controlled in that way. So based on how the locations are currently occupied, that row of the matrix is examined and the maximum value of that row is found. Whatever column that maximal value is in, the associated movement of players is used. Based on how well this movement works, the cell value is modified using Q-learning, a method of reinforcement learning. Case-based reasoning is used to load Q-table to be used by the system. If the Q-table has only had values decrease for a specific number of iterations, a new Q-table is loaded from the case base by comparing the team size, score, current location control and distance of each teammate to the various locations. Likewise, Q-tables are stored as new cases if they have gone a number of iterations without being replaced. This work identifies the growth of the case base size as a potential issue, but makes no attempt to deal with the increase.

Lastly, case-based reasoning has been used to retrieve game-plans in a real-time

strategy games [54]. Cases are generated by logging the game play of a human, who then expertly annotates the log file to combine each individual action into a structured plan. These plans can then be used to create cases, that are composed of the game state, a goal and the plan to achieve that goal given that game state. Each retrieved plan may require the retrieval of additional cases to solve each task in the plan, so there is a hierarchal link between cases that represents the complete game plan of the human expert.

### 3.2.2 Case-based Reasoning in RoboCup

Case-based reasoning has been applied in RoboCup in the simulation, small sized robot and four-legged robot leagues. Early theoretical work [57] in the simulation league, focused on creating a case representation and similarity measure that could be used to allow the team to execute specific set plays. This method converts the field into discrete regions and stores the number of players (both opponents and teammates) in each region. Along with the player positions, the cases also stores the time until a teammate is expected to control the ball, distance to other players and the distance to the ball. The distance calculation simply calculates the distance between the various features in the cases and sums these differences. Although they have provided a description of their approach, they do not present an implementation or perform any experimentation.

Similar theoretical work was performed in the four-legged league [27] to facilitate the execution of game plays. Each case is composed of the locations of the robots on the field as well as the level of possession (which team has the ball and how securely they possess it). In addition, they add meta-level features like the game score, game time, and how far a player is from opponents. The solution part of the case stores a game play to be executed by the team. As with the theoretical work in the simulated league, no implementation of this work is presented.

Ros and colleagues have also applied case-based reasoning to the RoboCup four-legged league. Their initial work [45, 48] focused on using case-based reasoning to identify situations when objects were in specific positions on the field. When these situations were encountered, set plays (moving the team in a certain manner) would be used. The cases they use contain the location of players, location of the ball, score of the game and time remaining in the game. As the solution portion, the case contains information about where each player on the team should move. The current state of the environment, where the robots are located, is compared to the case base and if any cases in the case base are similar enough (above a certain threshold) the team is then moved in a way defined by the most similar case. If no cases are similar enough, the players move in a random manner.

The other area Ros and colleagues looked into was multi-agent coordination [47] and cooperation [46]. In their previous work the case-based reasoning was performed by an external system that had a complete view of the field and could communicate with the players. In their multi-agent work they use a single player, called the coordinator, who has access to the case base and can instruct all of the members of the team. In order to avoid an incomplete world view, their current work relies on every player, including the opponents, to report their positions to the coordinator. This essentially leads to a similar situation as when the controlling was done by an external system with a complete world view. The major modification they introduce is that when calculating how similar the current environment is to the cases in the case base they attempt to find cases that can be transformed to the fastest, by changing the location of players. They draw on the idea that all teammates can move at that same time, in parallel, so the similarity is dependant on the furthest distance a single agent has to move. As well, if several cases are all found to be similar to the environment, the case with a solution that involves using the most teammates is selected. They argue that this encourages cooperation between agents.

Other approaches use techniques that give a noise-free, complete view of the field. In the simulation league, this is done by using a coaching agent. The coaching agent can see the position of everything on the field and can communicate with the players on its team. A coach has been used for guiding the movements of a player on the field [52] as well as predicting the behaviour of opponents [5]. Both of these approaches use higher level features to determine which feature weightings should be used for lower level features, and then use the lower level features to select an appropriate output.

In the small-sized league an external system has been used, like a coach, to control player movements and organize team formations [36]. This work examined three different tasks: controlling a goalie, selecting team formations and identifying game states. For goalie control, the location of objects on the defensives side of the field are contained in a case and the movement of the goalie is the solution portion of the case. In both the team formation control and when attempting to identify the game state, player positions as well as the location of the ball are stored within a case. The difference between the two tasks is the solution part of the case, with a team formation case storing what each player should do whereas the state identification cases store a label for an expertly defined game state.

While each of these approaches attempts to control the behaviour of a soccer playing agent or robot, none of them explicitly attempt to perform their learning through imitation.

### 3.2.3 Case-based Imitation

Case-based imitation aims to use case based reasoning to allow a software agent to imitate another agent and has currently been applied to the imitation of RoboCup soccer players in the RoboCup Simulation League [21, 31, 32]. This work will form the basis of this thesis, so special emphasis will be placed on describing it. In this

work, a case base is built by first logging all communication between the a soccer agent and the soccer server. This logged data contains the messages about what the player can see on the field, the objects visible to the player, and the actions the player performs, like kicking, dashing or turning. Each sensory message is matched with the actions that occur between it and the next sensory message. The sensory message is then parsed in order to determine the objects that are visible to the agent along with the position of those objects relative to the agent. A case is formed with the visible objects forming the case problem and the actions being the case solution. This allows an entire case base to be created in an automated manner through direct observation of an existing soccer agent.

The imitating agent can then use this case base during execution. During a game, the soccer server send the imitative agent a sensory message containing the objects that are visible to it, just like the real agent receives. The agent then compares the sensory information with the cases in the case base and selects an action to perform. A thorough description of the algorithms used to accomplish this is presented in Chapter 4.

Experimental results have shown that simple reactive agents can be imitated with a high degree of success but more complex agents, including the current state of the art RoboCup agents, can not currently be imitated [21,31,32]. These simple agents, when observed qualitatively, appear to behave in a manner similar to the original agents. In addition to testing data from various agents, tests have also examined varying the case representation or distance calculation [16, 31] and both manual [31, 32] and automated feature selection [21]. Using both manual and automated feature selection were found to improve the imitative ability, for all tested agents, compared to using all possible features. In addition, these improvements make it difficult to distinguish the imitative agent from the original agents during a game.

Thus far, no attempts have been made to optimize the case base used by an

imitative agent in order to deal with real-time constraints. While a large number of cases can be generated in an automated manner, the cases that are used during run-time are selected at random from the available cases. Thus, no attempt is made to select the best cases to use or reduce redundancy. Also, while feature selection has been employed no attempt has been made to remove unused features in order to reduce storage and computational costs.

### 3.2.4 Lessons Learned

A number of key lessons were learned relating to the application of case-based reasoning to RoboCup soccer:

- Existing approaches (excluding the case-based imitation work) use a simplified view of the world, where the locations of all objects are known at all times. This simplification makes both the storing and comparison of cases easier. Every case can contain the location of every object, so every case will have the same number of features. This makes the comparison of cases easier because there is no need to determine which features should be compared between cases. Consider two cases, the first with a single opponent visible and the second with two opponents visible. It is a non-trivial problem to determine which opponent in the second case should be compared to the opponent in the first case. By using a complete world model, that problem disappears, but in most actual applications a complete world model is not available.

- Outside of RoboCup, many of the CBR approaches make use of reinforcement learning. This is difficult when creating a general purpose imitator, since the goals of the teacher may not be known.

- In the work using Tetris, managing a large case base is examined. However, it

removes cases based on usage. This may not be appropriate if some cases are not used often but contain rare yet critical behaviour.

- RoboCup is a real-time domain, so all case-based reasoning must be performed in a fixed amount of time.

- Existing RoboCup approaches (excluding the case-based imitation work) use only a subset of the objects visible to the players to create a case. A human expert defines which objects should be included (usually other players and the ball) and all other objects are ignored. Although this may work for the given application, RoboCup, it does not lead to a general purpose case-based reasoning framework that can be applied to other domains.

- Other than in case-based imitation, in RoboCup all case bases are generated manually by a human expert. This can be a difficult task because it involves defining all necessary features of a case as well as the actions that are to be performed by the players. Creating large case bases in such a way can take a significant amount of time and effort.

- When generating a case base automatically, it becomes necessary to create a case base that is large enough to perform accurate imitation but small enough to meet the real-time restrictions of RoboCup.

## 3.3 Case Base Maintenance

In real-time domains, such as those discussed previously, the number of cases that can be contained within a case base are limited. This becomes a problem when there are more cases available than can be processed within the real-time limit. A subset of the cases must be selected to be used as the case base so that any real-time obligations

are met. This involves selecting which cases should be included in the case base such that the case base contains as much information as possible.

This field of research is known as case base maintenance, and is focused on refining the case base used by a case-based reasoning system [34]. These maintenance operations can generally be divided into three main groups: knowledge level, representation level and implementation level. Knowledge level operations directly modify the data contained in the case base, whereas representation level operations modify how cases are represented and implementation level operations deal with the structure of the case base [59].

### 3.3.1 Knowledge Level Maintenance

Early work in knowledge level maintenance examined a policy for deleting cases from a case base [50]. Initially, each case in the case base is examined in order to compute its coverage and reachability. The coverage of a case is the set of cases it can be used to solve and the reachability is the set of cases that can be used to solve it. Based on the coverage and reachability, cases are classified as either a pivotal, auxiliary, support or spanning case. A pivotal case is not reachable by any cases other than itself, whereas an auxiliary case is reachable by at least one other case with the same coverage as it, making it redundant. Support cases are found in groups and each have similar coverage. Removing an entire support case group would be like removing a pivotal case, however a subset can be removed without reducing the coverage. Spanning cases link two non-overlapping coverage regions.

Their deletion strategy involved removing cases in order of class. Auxiliary cases are removed first, followed by spanning cases, then support cases and finally pivotal cases. When deciding between cases of the same class, cases which are reachable by more other cases are deleted first. This strategy aims to remove cases that are solvable by the most other cases first while leaving cases that are not solvable by

other cases until the end.

Later work builds on this by introducing a relative coverage metric, rather than classifying the cases into groups [51]. This metric, calculated for each case, is used to identify cases that can solve cases that other cases can not solve. Each case in the case base contributes a value to each case that can be used to solve it, with the value contributed to each case being inversely proportional to the number of cases that can solve it. Cases then have their relative coverage computed by summing the contributions from all cases they can solve. Cases can then be ordered by their relative coverage value, with lower valued cases being deleted first. This guides the deletion process with a metric rather than based on classification.

The previous two methods have looked at maintenance from a case deletion standpoint, however other work has examined how coverage and reachability can be used to guide the addition of cases to a case base [63]. Using this approach, a new case base is created using cases from an existing case base. The goal is to have the new case base have the same coverage as the original but with fewer cases. Initially, the new case base is empty and all cases in the original case base have their coverage computed. Cases are iteratively added to the new case base, one at a time. Cases are selected for addition based on the increase to the coverage of the new case base that will result by their addition. This process terminates when the new case base has the same coverage as the original or a fixed number of cases have been added.

The previous three methods have all examined the maintenance process in terms of the coverage and reachability of the cases. Leake and Wilson [35] instead use a performance-based metric to guide maintenance. The coverage-based methods help to reduce a case base so that it can still be used to solve the same problems as the original case base, however this does not take into account the cost of adapting these cases to solve the problems. It may be useful to keep several cases with similar coverage, especially if these cases represent common problems, if it will reduce the

cost required to adapt these cases to input problems. Their metric, called relative performance, is computed for each case. For a given case, $C$, the relative performance looks at all cases that can be solved $C$. For each solvable case, $c_i$, the cost to adapt $C$ to solve $c_i$ is divided by the maximum cost to solve $c_i$, for all cases that can solve $c_i$. This value is then summed for all $c_i$ and produces the final relative performance metric. While this metric may sacrifice some coverage, it will include more cases in common problem areas to help reduce the overall time spent adapting solutions.

### 3.3.2 Representation and Implementation Level Maintenance

In the previous section on knowledge level maintenance, the contents of the case base were directly influenced through addition and deletion. This section will instead focus on maintenance techniques that do not directly modify the data but instead change how the data is represented or organized.

Aha and Breslow explore maintenance techniques for a conversational case-based reasoning (CCBR) system [3]. In CCBR, a user interacts with the system by providing additional information over a series of iterations. In this work, the user is presented with two lists: a list of questions and a list of solution cases. Each question is a request for the value of a specific feature. The user can then select a question to answer or finish the interaction by selecting a solution case. If a question is selected, the user then enters the answer to that question and both lists are updated based on the new information entered by the user.

Their maintenance operation involves three steps: indexing the case base, removing unnecessary features and ordering the importance of features. A top-down algorithm is used to induce a decision tree, with each node representing one or more cases. At each level, the feature that has a known value in the most cases is used

to split the cases. A branch is created for each value the feature takes as well as a separate branch for cases where the value is unknown. If no features can further divide the cases a node is created.

After this decision tree has been created, the path to each case is examined. Each case is modified so it only contains features used along its path in the decision tree, and the remaining features are then ranked based on the order they occur when moving down the tree. This maintenance approach not only changes the representation of cases by removing features but also performs an indexing of the features. This indexing of features is then used when updating the question list provided to the user, with questions related to higher ranked features appearing at the top of the list.

Case transformation and indexing has also been applied to unstructured textual cases [40]. The cases are not represented by a set of feature-value pairs but instead only contain text describing the problem and text describing the solution. Such unstructured cases make detecting inconsistencies and redundancies difficult. In order to overcome this, the cases are transformed into a set of keywords and phrases. Each of the keywords and phrases is then given an index representing what cases it appears in and how often it appears. This indexing makes it possible to search the case base for cases where both the problem and solution terms are similar in each case. Cases with similar problems and similar solutions can then be replaced by a single cases, reducing the redundancy.

Zhang and Yang use a three layered neural network to model the feature values, problems and solutions in a case base [61]. The first layer contains all possible values each feature can take, the second layer contains all possible problems and the third contains all possible solutions. The connections exist between each element in the feature value and problem layers, as well as between the problem and solution layers. These connections represent the weight that each element in the lower layer has on the layer above it. Whenever an input, represented as feature values, is given to

the system the most appropriate problems will be selected based on these weightings, which will in turn select the most appropriate solutions. If the user is not satisfied with the solution back propagation with occur to change the weights of each element. This approach is failure-driven and continuously updates how a case should be represented, in terms of feature values, based on user feedback.

### 3.3.3 Lessons Learned

Examining the literature of case base maintenance, there are several key aspects that would make the existing maintenance operations unsuitable for a case base used by an imitative agent. The following is a summary of those limitations:

- The knowledge level maintenance operations rely on cases that have known coverage and reachability. There are two major assumptions that these operations make. First, they assume that we know exactly which cases each case can solve. In domains like case-based imitation, where there are no rules to adapt a solution to a given problem, it may not be possible to accurately measure the coverage, reachability or adaptation cost of a case. Secondly, these approaches assume that the case base is a representative sample of the problem space. Since cases are generated through direct observation, the cases will only cover the problem space that has been encountered by the teacher during observation. If the teacher does not see a certain range of problems, the case base will not be a representative sample.

- In Aha and Breslow, the indexing method accepts features with unknown values but requires cases to be composed of a known set of feature-value pairs. This does not scale to cases in which each feature can have multiple values, like in case-based imitation where there can be multiple objects of the same type.

- In the indexing of unstructured textual cases, only the occurrence of each keyword or phrase is considered. This does not take into account other information such as the position of these items in a sentence. In case-based imitation, only comparing the number of objects in cases is not appropriate since the position of those objects plays an important role.

- While Zhang and Yang's neural network approach can continuously update the weights used during reasoning, it requires all inputs, problems and solutions to be known in advance. This is not a practical requirement in many domains, especially domains that look to minimizing the amount of expert knowledge required.

- Even if the case base is indexed in some way for faster retrieval, it is may still be useful to manage the size of the case base. Removing redundant cases would allow more informative cases to be added to the indexed case base. Also, the case base might need to fit into a limited amount of storage, like in a mobile robot, so removing redundant cases could help to reduce the storage requirements.

# Chapter 4

# Methodology

The research methodology that this thesis will follow has been used extensively in the field of case-based reasoning. More specifically, it has been used in literature that tests the effects of different algorithms on the case-based reasoning process ( [4], [58], [11], etc.) and in the field of case-based agent imitation [21, 32].

The steps of the methodology, with specific details for our application, are as follows:

1. *Build a case base*: The case base is automatically generated by observing the behaviour of a RoboCup soccer player (see Section 4.2). This will produce an initial, unprocessed case base.

2. *Apply preprocessing to case base*: The unprocessed case base will then have one, or more, of the preprocessing techniques applied to it. This will produce a preprocessed case base. Preprocessed case bases will be generated for each of the preprocessing algorithms, or combination of algorithms, being examined. This step of the methodology is the primary focus of this thesis.

3. *Examine the behaviour of the case-based imitation system using the preprocessed case bases*: Using the preprocessed case bases, as well as the initial unprocessed

case base, the various metrics will be collected through the use of a case-based imitation system (see section 4.3.1).

4. *Metric comparison*: The metrics for each algorithm will be compared to examine the effects of preprocessing (see section 4.3.2).

## 4.1 Unit of Analysis

### 4.1.1 Application Domain - RoboCup Simulation

The RoboCup Simulation league provides a suitable testbed to examine the methods described in this thesis due to the real-time constraints it imposes and the difficulty agents are presented in their world-view. Each player may only perform an action once per discrete time interval, called a cycle. If the player does not perform an action each cycle then it may be at a disadvantage compared to other players who act more often. This leads to a limit to the amount of computational time a CBR system can use per cycle.

A player can only see the objects that appear in its field of vision so at any given time it may only see a subset of the objects that exist in its environment. Also, due to noise associated with seeing objects, a RoboCup agent often does not possess enough information to properly differentiate similar objects. For example, it may only be able to tell that it can see a player, though not which particular player it sees. This uncertainty and limited world-view can pose interesting challenges when attempting to imitate an agent.

RoboCup is also a very active research domain, with numerous conferences, workshops and competitions devoted to it every year. As a result, data and software are often made freely available. This is especially useful in the simulated soccer league, since teams can be downloaded and used for experimental purposes.

### 4.1.2 Imitated Agents

For this thesis, we have selected several soccer agents with varying goals and complexity to use to test the preprocessing algorithms. Our intention is to vary the complexity of the tested agents so that we can identify how the algorithms perform for imitating different complexities of behaviour. Also, we include agents that do not attempt to play soccer. These agents are included to show the ability to imitate general behaviour and to show that the imitation techniques are not over-fitted to imitating soccer players.

The following five agents will be imitated:

1. **Sprinter**: runs from one goal net to the other goal net. This agent will run laps from one end of the soccer field to the other, completely ignoring the soccer game going on around it.

2. **Tracker**: this agent follows the ball around the soccer field. If the agent can not see the ball it turns until the ball enters its field of vision. The agent then runs towards the ball and stops when it is within a certain distance of the ball. As the ball changes position on the field, the agent will continue to follow the ball. Outside of simulated soccer we could see similar behaviour in a robotic assistant that follows a human around.

3. **Krislet** [33]: performs ball tracking behaviour similar to *Tracker* but kicks the ball toward the opponents goal when it gets near the ball. Unlike the previous two agents, *Krislet* will actively try to play soccer by scoring goals.

4. **NoSwarm**: this team is an extension of *Krislet* that behaves in the same manner as *Krislet*, except that a *NoSwarm* player will stay a fixed distance, $D$, away from the ball if it sees a teammate closer to the ball. Also, if the agent

is closer than distance $D$ to the ball, but a teammate is still closer to the ball, the agent will back up so that it is at least distance $D$ from the ball.

5. **CMUnited** [53]: a RoboCup Champion team which uses a layered learning architecture and a number of strategies including formation strategies. *CMUnited* players can have multiple states of behaviour and rely on inter-agent communication, making this team significantly more complex than the other teams that will be examined. We do not believe we will be able to successfully imitate this team since our approach does not currently handle multi-state behaviour or use non-visual stimuli like verbal communication. While we do not hope to successfully imitate this team, it provides a benchmark to help guide further research.

### 4.1.3 Algorithm Constants

The existing case-based imitation system, originally developed by previous students [31, 32] and latter improved in continued research [16, 20, 21], allows for a high degree of variability in both the algorithms used as well as the parameters for those algorithms. The system was designed in order to promote modular development such that algorithms used in various parts of the system could easily be changed. Although such a system is beneficial in that it is able to be extended and improved easily, it also requires an explicit description of what parameters will be used for experimentation and why such parameters were selected. In the following sections we will examine the various parameters and algorithms that are available in the case-based imitation system and identify those which will be used during our experiments.

### Nearest Neighbour Search

The core of the case-based imitation system is the k-nearest neighbour search. Looking at the k-nearest neighbour search algorithm we can identify two main areas of variability: the *number of neighbours* and the *distance calculation* used. The number of neighbours, or the $k$ value, represents the number of cases from the case base that will be found as possible solutions to the supplied problem (the input). In previous experiments, a value of $k = 1$ was found to be the optimum value to use [31]. This is due to the uneven distribution of actions in the case base. For example in RoboCup soccer, an unprocessed case base may typically have less then 0.1% of the cases representing the action of *kicking*, whereas the remainder of the cases will represent *dashing* and *turning*. As $k$ is increased we tend to see the more common actions (dashing or kicking) appearing among the retrieved neighbours. This can cause the case-based imitation system to select the most common actions simply because the majority of nearest neighbours are of that type, even though the 1-nearest neighbour may be of a different action type (and often the correct type).

### Distance Calculation Algorithm

Secondly, we have the various parameters related to the distance calculation. The pairwise distance between two cases is calculated as follows:

1. Features (i.e. objects that are visible on the field) from one case are matched with corresponding features from the second case. Algorithm 1 is an example of a matching algorithm. Any extra features, due to an unequal number of features in the two cases, are left unmatched.

2. The feature distance is then calculated between each matched pair of features. For each feature that did not have a match a penalty distance is applied.

3. Each of the feature distances and penalties are given a weighting based on the *type* of feature they are derived from (more detail on how these weights are derived is presented in Chapter 5).

4. All of the weighted distances and penalties are summed together and normalized to give the pairwise distance between the two cases.

This distance measure is presented in Algorithm 2. From this description we can identify three areas that can have variability: *how features are matched*, the *penalty applied to unmatched features* and the *weightings used*.

Two feature matching algorithms are currently in place in the system: *order index matching* and *Edmonds blossom shrinking algorithm* [14]. The order index matching algorithm (Algorithm 1) sorts the features (of each type) based on a particular attribute, in our case the distance of the feature from the agent, and features are matched with the feature of the same type in the other case that has the same positional index in the sorted list (e.g. the feature closest to the agent will be matched with the closest feature of the same type to the agent from the other case). This algorithm does not perform as accurate a matching as the blossom shrinking algorithm, but is requires far less computation. The reduced computation time of the order index matching algorithm was found to be significantly more beneficial than using the more precise blossom shrinking algorithm [32] and so we will experiment solely using the order index matching algorithm[1].

The penalty distance that is applied to unmatched features is a constant value and does not vary depending on the individual feature. Experimentally, it was found that applying a distance penalty of approximately *100* units produced the best results. As the penalty moves from this value (either higher or lower), the imitative ability of the system decreases. Although this may seem like an arbitrary penalty value, it is

[1]In more complex teams and domains it may be necessary to use a more precise matching, so we may again need to investigate more advanced matching algorithms.

**Algorithm 1** Order Index Matching

**Inputs:** firstCase, secondCase
**Outputs:** set of matched features (extra features are matched with null)

```
OIM(firstCase, secondCase)
   matches = { }
   types = all unique feature types in firstCase and secondCase
   for(each element, T, in types)
      list1 = all features of type T in firstCase
      list2 = all features of type T in secondCase
      sorted1 = list1 sorted by distance
      sorted2 = list2 sorted by distance
      N = max(sorted1.size(), sorted2.size())
      for(i =0; i<N; i++)
         if(i < sorted1.size() )
            m1 = sorted1.get(i)
         else
            m1 = null
         end if
         if(i < sorted2.size() )
            m2 = sorted2.get(i)
         else
            m2 = null
         end if
         newMatch = new Match(m1,m2)
         matches.add(newMatch)
      end loop
   end loop
   return matches
end
```

**Algorithm 2** Case Distance Calculation

**Inputs:** firstCase, secondCase
**Outputs:** distance between the two cases

```
distance(firstCase, secondCase, weights)
   distance = 0
   numDist = 0
   matches = match(firstCase, secondCase)
   for(each matched pair, P, in 'matches')
      type = P.getType()
      weight = getWeight(type)
      dist = pairwiseDistance(P)
      distance += dist*weight
      numDist ++
   end loop
   for(each unmatched pair, U, in 'matches')
      type = U.getType()
      weight = getWeight(type)
      dist = penalityDistance(U)
      distance += dist*weight
      numDist ++
   end loop
   distance = distance/numDist
   return distance
end
```

actual related to the size of the soccer field. The field has a length of *105* units, so a penalty of *100* units essentially implies that the feature is on the opposite side of the field as its matching feature.

The distances for each feature, or class of features, can have a weight applied to them. This weighting can be used to decrease the impact of certain features in the distance calculation process. Using non-equal weights for features has been shown to benefit the imitation process [21, 31], however our experiments will use equal feature weighting in order to remove the known benefits of feature weighting. This contains feature weighting to the chapters specifically related to feature weighting and removal (Chapter 5 and Chapter 8). This is done in order to attempt to identify if any preprocessing techniques actually work better with equal weights.

### Action Selection

After the k-nearest neighbour search has selected $k$ cases that are nearest to the given input problem, it then becomes necessary to select the action to perform. If more than one nearest neighbour is returned, when $k > 1$, the cases may have different actions associated with them so a method for selecting which action to use must be employed. Some examples of action selection techniques that are implemented in the case-based imitation system are:

- select the most common action amongst the neighbours' actions
- randomly select an action from the neighbours' actions
- perform a weighted vote, with closer neighbours getting a larger weight for their action

Although numerous action selection techniques exist to choose from, it is a trivial choice since we have previously made the decision to use a 1-nearest neighbour search.

All of the action selection methods will choose the action of the nearest neighbour, since that is the only available action to select from.

The more important choice, in our situation, is that of the *action estimation algorithm*. Each case in the case base can have several actions associated with it. This can be because the agent being imitated attempted to perform several actions in the same discrete time interval. This becomes an issue in some domains, RoboCup soccer specifically, where the agent is only allowed to perform one action per time interval. It then becomes necessary to estimate which action, if the actions differ, the agent meant to perform at that time interval. The action estimation could select the first action performed, last action performed or the most common action performed. We make the assumption that the agent being imitated has a certain amount of latency, so that the *last action performed* represents what action it wanted to perform given its current inputs and the previous actions were related to it still processing data from previous inputs.

### 4.1.4 Constant Summary

In summary, the following constant parameters will be used in the case-based imitation system:

- **Nearest neighbours** : 1-nearest neighbour search
- **Matching algorithm** : order index matching
- **Unmatched penalty** : 100 units
- **Feature weights** : all equally weighted (unless noted in the experiment)
- **Action selection** : action associated with the nearest neighbour
- **Action estimation** : last action performed

### 4.1.5 Experimental Parameters

Our experiments will vary parameters (algorithms and data used) in order to address the following research questions:

- *In the domain of simulated RoboCup soccer, how can the imitative ability of a real-time case-based imitation agent improve by preprocessing the case base it uses?*: This question will be further subdivided based on the preprocessing techniques examined.
  - *Will removing less relevant features from cases result in a decrease in execution time, and by how much, without negatively affecting the imitative performance of a RoboCup imitation agent?*: The feature selection algorithm used will be varied (Chapter 5) and the results will be compared with a case base that includes all possible features.
  - *Can sets of similar cases, grouped using data clustering, be replaced with a single prototypical case without a significant negative effect on the performance of a RoboCup imitation agent? What level of compression can be achieved before negative performance occurs?*: Algorithms for clustering cases (Chapter 6) and methods for replacing a set of cases into a single prototypical case (Chapter 7) will be examined.
  - *Does combining feature removal and prototyping result in improved performance compared to using each technique separately, and does the order of preprocessing matter?*: The most promising feature selection and prototyping methods will be combined (Chapter 8) to see if there is any improvement in using both techniques.
- *Do the preprocessing techniques work when imitating a variety of teams?*: Our goal is to identify techniques that are applicable to any spatially-aware software

agent we wish to imitate. Each experiment will be conducted using data from five RoboCup agents (Section 4.1.2), each with different goals and behaviour.

### 4.1.6 Summary of Parameters

The following summarizes the parameters that will be varied during experimentation. It should be noted that not all combinations of these parameters will be used. For each method examined (feature reduction, clustering, prototyping and hybrid) only the parameters that performed best will be used in subsequent chapters. For example, only the best feature reduction algorithm will be used in the hybrid experiments.

- **Teams:** Sprinter, Tracker, Krislet, NoSwarm and CMUnited
- **Maximum size of case base:** computed for each team and feature weighting
- **Feature reduction - algorithms:** None, backward sequential search, genetic algorithm, backward sequential search with dynamic training set, genetic algorithm with dynamic training set (see Chapter 5)
- **Clustering - algorithms:** None, leader algorithm, single-linkage algorithm, distance vector with k-means algorithm (see Chapter 6)
- **Clustering - initial sizes of case bases before clustering:** 6000, 12000, 18000, 24000
- **Prototyping - algorithms:** None, cluster member, average case, range case (see Chapter 7)
- **Prototyping - dirty clusters:** Prototype dirty clusters, leave dirty clusters unchanged (see Chapter 7)
- **Prototyping - initial size of case base before clustering and prototyping:** 6000, 12000, 18000

- **Hybrid - order of preprocessing:** Feature removal followed by prototyping, prototyping followed by feature removal (see Chapter 8)

## 4.2 Data Collection

The data collection process involves building case bases that contain cases that represent an agent's reaction to its environment. In the RoboCup context, this involves creating cases that store what the agent can see, at a specific moment in time, and the resulting action the agent performs. The case bases are built in an automated manner using a tool [37] that acts as a proxy between the RoboCup player that is to be imitated and the RoboCup soccer server [43]. Instead of connecting directly to the server, the player connects to the proxy utility and the proxy utility connects to the server. This places the utility directly between the player and server. All communication between the player and server is logged before it is forwarded along (Figure 4.1). The logged data can then be converted into a case base by mapping server messages (representing what the agent can see) with agent messages (the actions the agent performs) that occur directly after them.

For each of the five types of agents being examined (Section 4.1.2), logs were generated by observing a player of that agent type playing 24 complete soccer games. Each game involved 11 players (the maximum allowable size of a soccer team) of one type of agent playing against 11 opposing players. Krislet was selected as the opposing players, for all five types of agents being logged, in order to keep consistency. Krislet was selected as the opponents because they actively play soccer, unlike Sprinter or Tracker, and they provide the other team an opportunity to possess the ball during the game. This is in contrast to a team like CMUnited who might be so skilled that the other team would never have a chance to kick the ball. In summary, the data collection constants are:

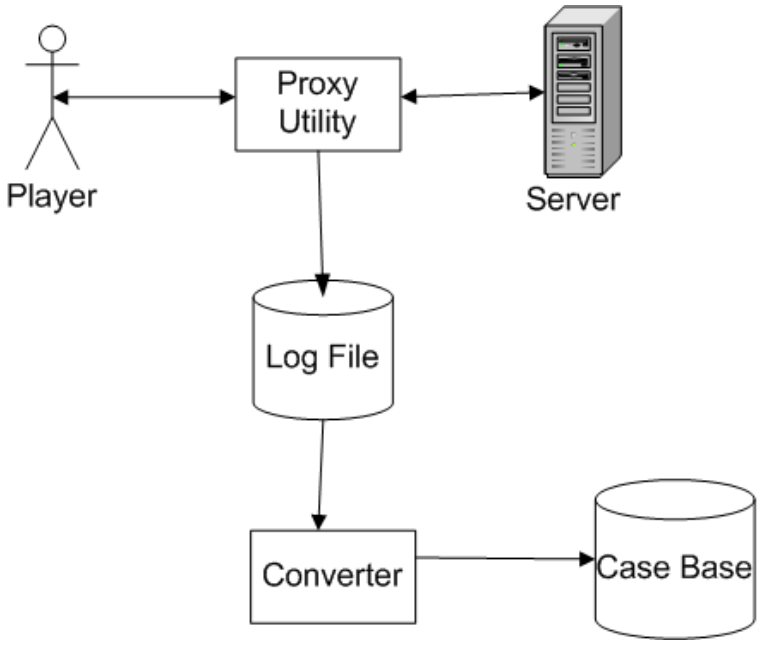


**Figure 4.1:** Building a case base using a proxy utility.

- **Collected logs:** 24 per team
- **Team size:** 11 per team
- **Opponent team:** Krislet

## 4.3 Data Analysis

For each agent, the 24 log files that were generated in Section 4.2 will be used to create a single case base[2]. These case bases, referred to as the *complete sample case base* for the particular agent, will contain all of the observed cases for the agents and will be used to create smaller case bases for testing and training. The training and testing case bases will be created by randomly selecting cases from the complete sample case bases.

[2]Depending on the agent, these complete case bases range in size from approximately 95000 to 125000 cases

- **Training case base**: Different experiments may use different sized case bases. For an experiment requiring a training case base of size $N$, a training case base will contain $N$ cases randomly selected from the complete sample case base for the agent being tested.

- **Testing case base**: For each experiment *1500* cases will be randomly selected from the complete sample case base for the agent being tested. This value was selected arbitrarily and represents approximately 25% of the cases an agent might encounter during an actual soccer game. Each of these testing cases is used as input to the case-based reasoning system and the output action of the case-based reasoning system is compared to the action stored in the test case. These comparisons are used to compute the various metrics that measure the performance of the imitation system.

Testing methods that randomly select a single case base and then use subsamples of that case base for training and testing (N-fold cross validation, leave-N-out validation, etc.) were not used because some of the preprocessing techniques attempt to reduce redundancy. For example, the prototyping techniques presented in Chapter 7 attempt to merge similar cases into a single prototypical case. Therefore, after prototyping a case base we would assume that each remaining case is somewhat dissimilar to all other remaining cases. If the prototyped case base was then subsampled into a training and testing case base we would not expect any of the testing cases to have similar cases in the training set, so the nearest neighbour algorithm would only find dissimilar neighbours.

Each experiment will be performed 25 times and the average values of the computed metrics will be presented. The experiments will then be compared to identify any statistically significant differences in the results.

### 4.3.1 Metrics

The metrics collected during experimentation are calculated are as follows:

- **Execution time**: This is the mean amount of time it takes to select an action to perform after receiving an input (the state of the environment). This measures how long the case-based reasoning process took to complete.

$$ET_{MEAN} = \frac{\sum_{i=1}^{n} ET_i}{n} \tag{4.1}$$

  Where $ET_i$ is the execution time to select an action when given the $i^{th}$ test case, and $n$ is the total number of test cases that are processed.

- **Accuracy**: This is the number of times the action in the test case matched the action returned by the CBR system ($c$) divided by the total number of test cases the CBR system was given ($n$). Accuracy values range from 0 (low) to 1 (high).

$$Accuracy = \frac{c}{n} \tag{4.2}$$

  For the value $c$, correctly matching actions are actions of the same type. This does not take into account the similarity of the parameters of an action, like the power of a kick, but just if the actions were of the same type. While a matching that also takes parameters into account might be more stringent, this method helps to identify when the returned action is close to the desired action. If the action is at least of the correct type, it might later be possible to modify it slightly to get the correct parameters.

  Unfortunately, the accuracy metric only measures how the system is performing in predicting all types of actions. It gives us no information related to if one

type of action is being predicted more (or less) accurately than other actions or if a certain type of action is often selected incorrectly. If one class of action occurred a disproportional number of times in the data, then simply selecting the most common class of action could lead to a high overall accuracy but would perform poorly for the remaining classes.

- **Recall**: Recall is calculated for each of the $j$ actions that the agent can perform. It is the number of times the action was correctly selected ($c_j$) divided by the number of times the action should have been selected ($n_j$). This is essentially the accuracy of each individual action. Recall values range from 0 (low) to 1 (high).

$$Recall_j = \frac{c_j}{n_j} \tag{4.3}$$

- **Precision**: As with the recall, the precision is calculated for each of the $j$ actions that the agent can perform. It is the number of times the action was correctly selected ($c_j$) divided by the number of times the action was selected ($w_j$). This statistic will show if an action is often being selected incorrectly. Precision values range from 0 (low) to 1 (high).

$$Precision_j = \frac{c_j}{w_j} \tag{4.4}$$

- **f-measure**: The f-measure combines the recall and precision values into a single metric. As with precision and recall, values range from 0 (low) to 1 (high).

$$F_j = \frac{2 * Precision_j * Recall_j}{Precision_j + Recall_j} \tag{4.5}$$

If we combine the f-measure values for all $j$ actions we get the global f-measure

value.

$$F_{global} = \frac{1}{j} \sum_{i=1}^{j} F_j \tag{4.6}$$

For all experiments performed, the global f-measure value will be the primary metric used for comparison.

### 4.3.2 Parameter Comparison

This thesis examines a variety of algorithms and preprocessing techniques, many of which are alternatives to each other. The experiments that are performed vary the algorithm used for a specific task while keeping all other parameters constant. The computed metrics must then be compared in order to select which algorithms are most beneficial to the case-based imitation system. The following comparisons will be performed:

- Comparing each preprocessing algorithm to using no preprocessing. This will identify if the preprocessing technique is an improvement over a non-preprocessed case base. A t-test will be used to see if there is a statistically significant difference between metrics.

- Comparing preprocessing algorithms used for the same task. This will identify which algorithm is best suited to perform the specific preprocessing task.

# Chapter 5

# Feature Reduction

In case-based reasoning, an input problem is compared to the cases stored in the case base and the most similar cases are used to select the solution to the input problem. So, for each supplied problem, the case-based reasoning system must make numerous comparisons between the problem and cases in the case base. These comparisons, either similarity or dissimilarity measurements, can result in a significant portion of the computational time required by the CBR system due to the number of comparisons required.

If a case, $C$, is composed of a set of $n$ features, $f_1 \ldots f_n$, then any comparison between cases will require a comparison between the $n$ features that make up each case[1]. In the RoboCup domain, the features that compose a case are the objects visible to an agent. As the number of features in a case increases so to does the number of feature comparisons necessary when performing a comparison between two cases. In the RoboCup example, more features means more objects are visible to the agent so the agent has more visible objects it needs to process. If the number of features in a case can be reduced, either by removing some features from the case or using a comparison calculation that ignores some features, then the computational

[1]If the comparison only compared a subset of the case features then it would have a bias toward those features and would already be performing a form of feature selection

time required by the comparison computation will be reduced. As a byproduct, the computational time required by the case-based reasoning system to solve a problem will also be reduced since each case comparison will execute in less time.

Reducing the number of features in a case not only reduces the time required to compare cases, but it can also improve the performance of the case-based reasoning system. The cases may contain features that are either irrelevant or redundant. In an imitation context, irrelevant features might occur if the agent being imitated does not reason with all of the objects that it can see, but only a subset of those objects. If these features are included in the comparison between cases they may actually reduce performance. This is especially true when using the k-nearest neighbour algorithm, or variants thereof, since k-nearest neighbour algorithms have been found to be sensitive to redundant and irrelevant features [25]. Since many case-based reasoning systems use k-nearest neighbour algorithms [1], specifically case-based imitation systems [21, 31, 32], this issue becomes necessary to take into account during preprocessing. Also, if each comparison takes less time then more comparisons can be made in the same amount of time, allowing a larger case base size.

## 5.1 Feature Selection Methods

Feature selection algorithms can be divided into two primary groups: *wrappers* and *filters*. Wrapper algorithms [28] search for an optimal feature weighting by evaluating the weightings when using them in a specific target algorithm. For example, the wrapper algorithm would evaluate potential weighting by using those weights in the CBR system. This is in contrast to filter algorithms [26] that select features without using the target algorithm, instead using some other sort of metric to evaluate weights. Wrapper algorithms are often favoured because they directly use the algorithm that will use the feature weights to evaluate the weights, avoiding the bias filter algorithms

impose when using other metrics. The primary drawback of wrapper algorithms is that using the target algorithm to evaluate weights can lead to a higher computational cost. In order to avoid the bias and need to define a weight evaluation metric associated with filter algorithms, we will focus exclusively on wrapper algorithms.

### 5.1.1 Binary Weight Feature Selection

The comparison of two cases, either their similarity or dissimilarity, can be thought of as a function of the similarity or dissimilarity of the features in those cases. For example, if two cases, $C_1$ and $C_2$, each have $n$ feature then the dissimilarity, $D$, between the cases is a function of the dissimilarity, $d_i$, between each of the features:

$$D(C_1, C_2) = f(d_1, \ldots, d_n)$$

By giving some features a weighting of zero, those features can be removed from the function and therefore no longer have any impact on the dissimilarity between cases[2].

In binary feature weighting, each feature is given a weight of either *1* or *0*. A weight of *1* means that the feature should be kept, while a weight of *0* means that the feature should be excluded. While a binary approach is limited in that each feature can only have two possible weights, it often produces more general weights that are less susceptible to overfitting [41] compared to continuous weighting.

We will examine one such binary feature selection algorithm, the backward sequential search (BSS) algorithm [2]. The BSS algorithm requires two parameters. The first parameter is the minimum percentage a feature set must improve over the current best feature set in order to become the new best feature set. The second parameter is the number of feature sets that are examined, without finding a new

[2]The same is valid for similarity measures as well

best feature set, before the algorithm terminates. During execution, the BSS algorithm maintains two lists: open feature sets and closed feature sets. For a feature set to be open, it has had its performance evaluated but has not been evaluated to see what *children* it has. This algorithm performs a backwards search starting from the feature set that contains all possible features (all features have a weight of *1*). For a given feature set, its children are all the possible feature sets that can be generated by removing a single feature from the feature set (setting one of the *1* weighted features to a *0* weighted feature). At each iteration, the feature set in the open list with the highest performance evaluation has its children determined. If any of the children do not exist in either the open or closed lists, their performance is evaluated and they are added to the open list. The parent is then added to the closed list. The algorithm terminates when there are no more feature sets in the open list or after a number (provided as input to the algorithm) of feature sets are examined without finding a new best feature set. The algorithm is shown in Algorithm 3.

### 5.1.2 Continuous Weight Feature Selection

While binary feature weights can only take two possible values, *0* or *1*, continuous weights can take those values as well as any values between them. This allows for features to not just be included or excluded, but they can also be given weights related to their importance. In binary weighting every included feature is found to be equally important because they have the same weight of *1*. However, in continuous weighting features can be included but have different weightings, with weights closer to *1* representing more important features and weights closer to *0* representing less important features. Such a weighting scheme can allow for more finely tuned weights but does leave open the possibility of overfitting as was described in Section 5.1.1.

One such method for continuous feature weighting is to use a genetic algorithm. A genetic algorithm attempts to mimic the process of biological evolution in order

**Algorithm 3** Backward Sequential Search - Binary Wrapper Algorithm

**Inputs:** minIncrease, maxNonImproving, targetAlgorithm
**Outputs:** currentBest: best weights found

```
BSS(minIncrease, maxNonImproving, targetAlgorithm)
    currentNonImproving = 0
    allFeatureSet = set where all features have non-zero weight
    openList = {allFeaturesSet}
    closedList = { }
    currentBest = null

    while (currentNonImproving < maxNonImproving)
       bestOpen = remove feature set from openList with highest evaluation
       children = determineChildren(bestOpen)
       for(each child C in children)
          if(C does not exist in openList or closeList)
             targetAlgorithm.evaluate(C)
             openList.add(C)
          end loop
       end loop
       if (bestOpen.evaluation > currentBest.evaluation + minIncrease)
          currentBest = bestOpen
          currentNonImproving = 0
       end loop
       else
          currentNonImproving ++
       end loop
       closedList.add(bestOpen)
    end loop
    return currentBest
end
```

to evolve the optimum solution to a problem [29]. In feature weighting, the genetic algorithm is used to evolve a set of weightings into the optimum weighting for the problem. Initially the genetic algorithm randomly creates a set of weightings and evaluates the weightings using a *fitness function* (in our case the CBR algorithm). This set of weightings is called the *population* and is of a fixed size, the *population size*. The algorithm then performs a number of iterations called *generations*.

Each generation, the genetic algorithm selects the fittest weights, called the *parents*, in the population to use to create new weights. The parents are then used to create variations of themselves, called the *children*, who are then evaluated using the fitness function and added to the population. The population then contains the population from the beginning of the generation as well as the newly created children, so the least fit individuals in the population are removed so that the population returns to the fixed population size. The algorithm terminates after a fixed number of generations[3] and the member of the current population with the highest fitness is the best weighting found by the genetic algorithm. The pseudocode is shown in Algorithm 4.

### 5.1.3 Dynamic Training Set Feature Selection

The use of feature selection to improve the imitative ability of a software agent has been shown to be successful, using both manual feature selection [32] and feature selection algorithms [21]. However, in order to address software agents with real-time concerns we also need to look at feature selection as a tool for reducing the computational costs of a case-based reasoning system.

As we described at the start of this chapter, the execution time required to solve a problem using a case-based reasoning system is related to the number of features that are used in the comparison between cases. Given the total time to solve a problem

[3]Alternatively, some implementations terminate when one of the weightings achieves a target fitness level. This generally is used when the fitness value has a known upper bound

**Algorithm 4** Genetic Algorithm - Continuous Wrapper Algorithm

**Inputs:** numGenerations, populationSize, targetAlgorithm
**Outputs:** best weighting found

```
GA(numGenerations, populationSize, targetAlgorithm)
    population = randomly create ‘populationSize’ weightings
    for(each weighting W in population)
       targetAlgorithm.evaluate(W)
    end loop
    for(i = 1 ; i < numGenerations ; i++)
       parents = selectParents(population)
       offspring = createChildren(parents)
       for(each weight W in children)
          targetAlgorithm.evaluate(W)
          population.add(W)
       end loop
       while( population.size() > populationSize)
          leastFit = weighting in population with lowest evaluation
          population.remove(leastFit)
       end loop
    end loop
    return weighting in population with highest evaluation
end
```

case using a CBR system, $t_{tot}$, when the CBR system uses a case base of size $N$[4], then the average execution time cost per case in the case base, $t_{case}$, is:

$$t_{case} = \frac{t_{tot}}{N} \tag{5.1}$$

This makes the assumption that the execution time increases in a linear manner as the number of cases in the case base increases. If each case is composed of $n$ types of features[5], then the average execution time cost per feature type, $t_{feat}$, is:

$$t_{feat} = \frac{t_{case}}{n} \tag{5.2}$$

It should be noted that this assumes that each type of feature has an equal execution cost. If this is not the situation, each feature type can be assigned a unique cost value.

After applying a feature selection algorithm, we may have features that should be excluded (given a weight of *0*). If we originally had $n$ features but now only wish to include $m$ features, where $m < n$, then we can recalculate our estimated cost per case, $t_{casenew}$, and total time to solve a problem using CBR, $t_{totnew}$, as follows:

$$t_{casenew} = t_{feat} * m \tag{5.3}$$

$$t_{totnew} = t_{casenew} * N \tag{5.4}$$

Since $m < n$, we know that $t_{casenew} < t_{case}$ and $t_{totnew} < t_{tot}$. The removal of

[4]This assumes that every case in the case base is compared to the problem case. If the case base is structured differently, such as a hierarchal layout, then $N$ would be the average number of cases that are compared to the problem case.

[5]This assumes that objects can not be uniquely identified, but only what type of object is known. For example, a soccer player is known to be of type *player*, but has no unique identifier such as a name.

features leads to a decrease in the cost of each comparison between cases as well as a decrease in the total time required to search the case base for a solution to the input problem. If the case base search must be completed within a real-time limit, $t_{limit}$, then the removal of features will allow more cases to be placed in the case base. The number of cases in the case base can be changed so that $t_{totnew} \approx t_{limit}$.

The downside of existing wrapper algorithms, when taking into account the real-time concerns, is that they select the features that will optimize the performance of a given algorithm when using a fixed-sized training sample. One should note that performing feature selection on a fixed-sized training sample will produce an optimum feature weighting for that training sample and will not take into account that every feature removed will result in more cases that can be evaluated within the real-time limit. For example, the removal of a feature might not improve the performance of the target algorithm using a fixed-sized training set but the performance might be increased if that feature was removed so that more cases could be added to the training set (potentially improving the diversity of the training set).

To overcome this limitation we propose using a dynamic-sized case base as the training set. For each potential weighting that is evaluated by a wrapper algorithm, rather than using a fixed-sized training set for all weightings a training set size will be estimated based on the particular weighting. Based on the number of non-zero feature weights, $m$, in the particular weighting the estimated time to search a case base of size $N$ can be calculated using Equation 5.3 and Equation 5.4. If it takes an amount of time, $t_{totnew}$, to perform CBR using a case base of size $N$, then we can estimate the number of cases, $N_{max}$, that can be used within our real-time limit, $t_{limit}$, as:

$$N_{max} \approx \frac{t_{limit} * N}{t_{totnew}} \tag{5.5}$$

The primary benefit of this approach is that it allows wrapper algorithms to evaluate a weighting based on the number of cases that such a weighting can potentially use in a case base. It becomes possible to evaluate whether it is better to keep a feature or remove it in order to increase the case base size and potentially increase the diversity of the case base. Also, this technique has the benefit of being usable by any wrapper algorithm. The dynamic training set algorithm (Algorithm 5) is initially constructed by supplying a large collection of cases, the real-time limit that must be enforced and the evaluation algorithm it will wrap around. It can then be used in place of the evaluation algorithm in the feature selection algorithm. During execution of the feature selection algorithm, the *evaluation* method will be invoked in the dynamic training set algorithm which will in turn call the *evaluation* method from the evaluation algorithm.

**Algorithm 5** Dynamic Training Set

**Method:** Evaluate a weighting
**Inputs:** weighting
**Outputs:** evaluation of weight

```
//first constructed by supplying the target algorithm (algorithm),
//  real-time limit (limit) and list of cases (caseList)

evaluate(weighting)
    caseCost = 0
    for(each non-zero weight in weighting)
       caseCost += execution time cost of the feature being weighed
    end loop
    estimatedSize = limit/caseCost
    trainingCaseBase = randomly select 'estimatedSize' cases from caseList
    algorithm.setTrainingSet(trainingCaseBase)
    performance = algorithm.evaluate(weighting)
    return performance
end
```

## 5.2 Experimental Results

Our experiments in this chapter aim to address the following issues:

- Establishing the real-time limit for the system
- Determining the maximum number of cases that can be examined within the real-time limit
- Validating our assumption that the execution time increases linearly as the case base size increases
- Determining the performance when all features are included
- Determining the performance when using both a binary and continuous feature selection algorithm
- Determining the performance when using dynamic training set size

### 5.2.1 Equal Weight Results

In the RoboCup Simulation League, each player may only perform an action once per discrete time interval, called a cycle. If the player does not perform an action each cycle then it may be at a disadvantage compared to other players who act more often. The entire process (Figure 5.1) of identifying what objects are currently visible to the agent, using CBR to select the appropriate actions to perform and performing those actions should then be completed within one cycle (of length 100ms [12]). Also, given that the CBR process is not the only task the agent needs to complete within each cycle, we will set our time limit for performing CBR to half of the cycle (50ms).

Now that a time limit has been established for the case-based reasoning performed by an imitative agent, we can determine how many cases the CBR system can process within the time limit. For these experiments, the average amount of time it took to

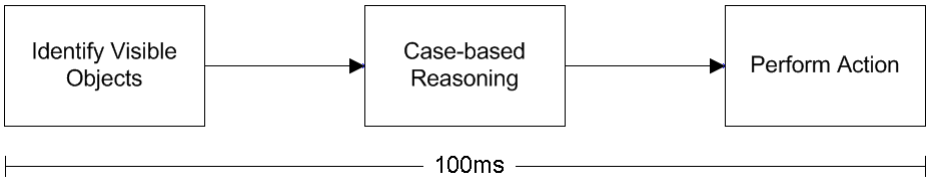


**Figure 5.1:** The activities the agent must perform in one 100ms cycle.

find a solution to an input problem was measured when using case bases of various sizes (with all features included). For Sprinter, Tracker, Krislet, NoSwarm and CMUnited (Figure 5.2) we can see that the time increases in a nearly linear manner as the size of the case base increases. This verifies the assumption that was made in Section 5.1.3. Applying linear interpolation on the data used to generate Figure 5.2, we can determine the maximum case base size that can be searched within the 50ms time limit (Table 5.1).

In is interesting to note that there is a difference in the maximum case base size for each team. This is related to the teams behaviour and how they view the field. The cases contain a representation of the field of vision of the agent, so depending on how they observe the field they will have a different number of objects in their field of vision. For example, Tracker, Krislet and NoSwarm all have similar maximums because they behave in a similar manner and therefore see similar configurations of objects on the field. Each of these teams try to stay facing the ball, where other objects are likely be located in a game of soccer, leading to more objects in each case. This is in contrast to a Sprinter agent that pays no attention to the ball and may often be looking away from the action on the field. It may seem counterintuitive that CMUnited, the most complex team, has a higher maximum number of cases than Tracker, Krislet and NoSwarm. This is because a CMUnited agent will often look around the field, often to areas with few objects, rather than focusing on areas of high object concentration.

In order to establish a baseline for comparing further experiments, the performance

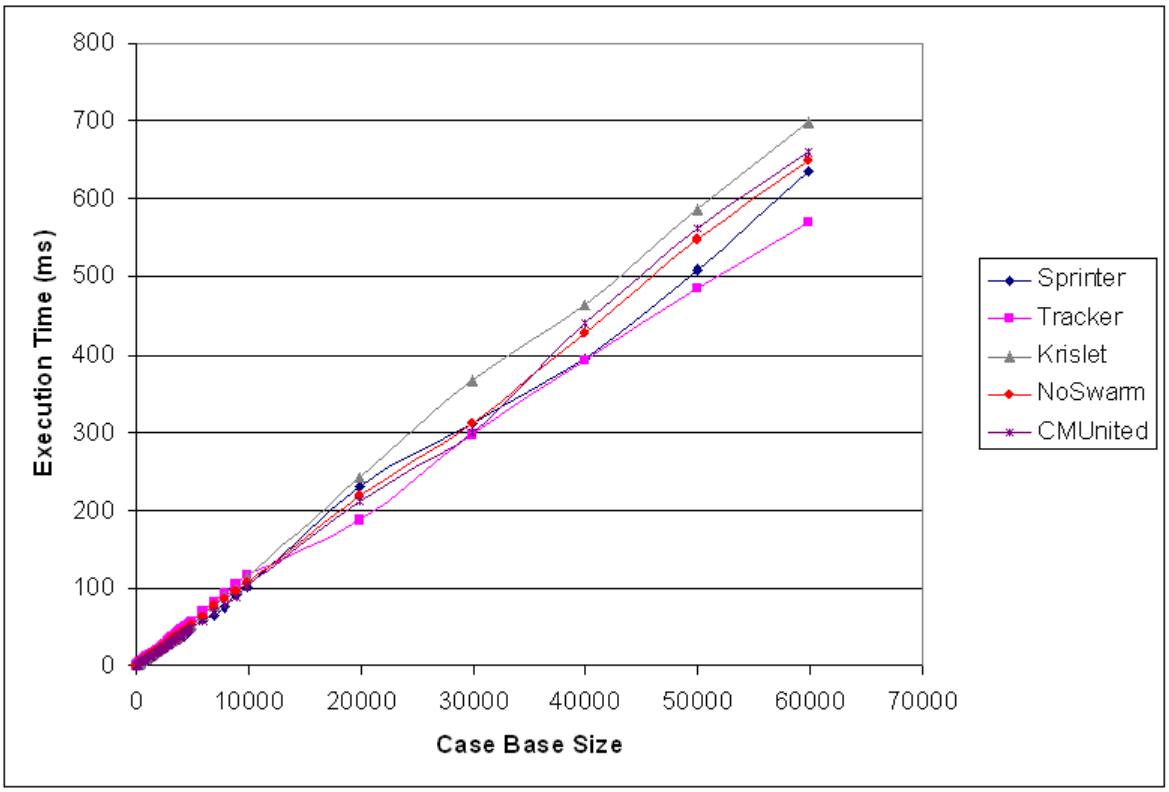


**Figure 5.2:** Time to search the case base as case base size increases

for each agent was gathered when all features were included and the case-based reasoning system used the maximum sized case bases (which are given in Table 5.1). A summary of those results, when all features are included, can be found in Table 5.2. Also, the performance of the system was evaluated when case-based reasoning was not used and actions were selected at random. The results using random action selection are found in Table 5.3 and show that the case-based reasoning approach provides a significant increase over the random action selection. It should be noted that the results, using random action selection, appear higher for Sprinter and Tracker but that is because those agents only use two actions instead of three.

| | **Maximum Size** |
|---|---|
| **Sprinter** | 5271 +/- 9 |
| **Tracker** | 4409 +/- 6 |
| **Krislet** | 4612 +/- 7 |
| **NoSwarm** | 4727 +/- 6 |
| **CMUnited** | 5078 +/- 9 |

**Table 5.1:** The maximum case base size that can be searched in 50ms.

**Table 5.2:** Performance using all features and a maximum sized case base.

| | **Cases** | $f1_{global}$ | **Accuracy** | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|---|
| **Sprinter** | 5271 | 0.69 +/- 0.004 | 0.84 | — | 0.91 | 0.48 |
| **Tracker** | 4409 | 0.77 +/- 0.003 | 0.77 | — | 0.76 | 0.78 |
| **Krislet** | 4612 | 0.58 +/- 0.004 | 0.76 | 0.25 | 0.70 | 0.80 |
| **NoSwarm** | 4727 | 0.61 +/- 0.003 | 0.81 | 0.20 | 0.81 | 0.82 |
| **CMUnited** | 5078 | 0.59 +/- 0.002 | 0.65 | 0.48 | 0.71 | 0.58 |

### 5.2.2 Binary Weight Results

Our next experiments examine using the backward sequential search algorithm to select the features to be used by the case-based reasoning algorithm. The BSS algorithm terminated after *5* iterations without finding a new best weighting, and a weighting had to be at least *0.1%* better than the previous best to become the new best weighting. The case-based reasoning algorithm used to evaluate the weightings used a fixed-size case base. A summary of the performance using the best weightings found by the BSS algorithm can be found in Table 5.4. The experiments were conducted *25* times for each team and for some teams different experiments resulted in different weightings being found. The number of times each feature was found to

**Table 5.3:** Performance using random action selection.

| | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|
| **Sprinter** | 0.51 +/- 0.009 | — | 0.50 | 0.52 |
| **Tracker** | 0.47 +/- 0.007 | — | 0.48 | 0.46 |
| **Krislet** | 0.31 +/- 0.010 | 0.32 | 0.33 | 0.28 |
| **NoSwarm** | 0.31 +/- 0.008 | 0.30 | 0.32 | 0.31 |
| **CMUnited** | 0.33 +/- 0.012 | 0.32 | 0.33 | 0.34 |

have a non-zero weighting is presented in Table 5.5, with a value of *25* indicating the feature was found to have a non-zero weight in all tests. Values of the global f-measure marked with (*) denote a statistically significant increase over the global f-measure results when using all features (p=0.01).

**Table 5.4:** Performance using weights from backward sequential search.

| | **Cases** | $f1_{global}$ | **Accuracy** | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|---|
| **Sprinter** | 5271 | 0.89(*) +/- 0.002 | 0.95 | — | 0.97 | 0.82 |
| **Tracker** | 4409 | 0.97(*) +/- 0.003 | 0.97 | — | 0.97 | 0.97 |
| **Krislet** | 4612 | 0.83(*) +/- 0.002 | 0.98 | 0.53 | 0.98 | 0.98 |
| **NoSwarm** | 4727 | 0.76(*) +/- 0.004 | 0.94 | 0.40 | 0.94 | 0.95 |
| **CMUnited** | 5078 | 0.60(*) +/- 0.002 | 0.62 | 0.56 | 0.66 | 0.59 |

**Table 5.5:** Number of times a feature had a non-zero weight with backward sequential search.

| | **Ball** | **Goal** | **Line** | **Flag** | **Teammate** | **Opponent** | **Unknown Player** |
|---|---|---|---|---|---|---|---|
| **Sprinter** | 0 | 25 | 0 | 0 | 0 | 0 | 0 |
| **Tracker** | 25 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Krislet** | 25 | 0 | 0 | 0 | 0 | 0 | 0 |
| **NoSwarm** | 25 | 0 | 0 | 0 | 0 | 0 | 0 |
| **CMUnited** | 23 | 8 | 5 | 17 | 6 | 2 | 10 |


**Figure 5.3:** A graphical representation of the chromosome used by the genetic algorithm

### 5.2.3 Continuous Weight Results

Each potential weighting, called a *chromosome*, is composed of *genes* that each store the weight for one type of object. In these experiments, every chromosome is composed of seven genes since there are seven weights being used. A graphical representation of a chromosome can be seen in Figure 5.3, with the letters representing the weight for the particular object type (ball, goal, line, flag, teammate, opponent, unknown player).

The genetic algorithm we will use for feature selection[6] will have a population size of *50* and will run for *1000* generations. The initial population of chromosomes is initialized with each gene containing a random weight. At each generation, the parents will be selected by *tournament selection* with a tournament size of *4*. Tournament

[6]The Java Genetic Algorithms Package (JGAP) was used to implement the genetic algorithm and is available from http://jgap.sourceforge.net

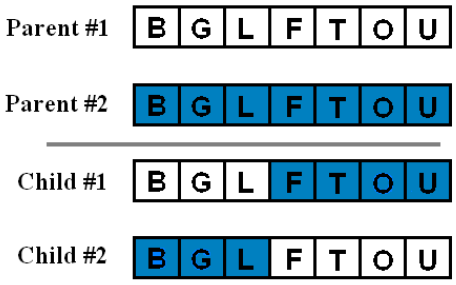


**Figure 5.4:** The crossover operation

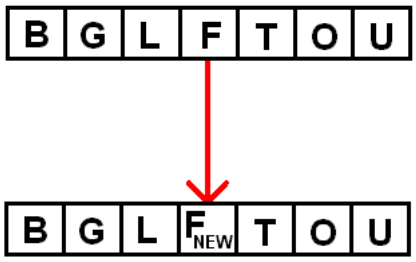


**Figure 5.5:** The mutation operation

selection works by randomly selecting several members of the population, in our case *4*, and then from those selecting the one with the highest evaluation to be used as a parent (70% of the time) or one of them at random (30% of the time). This tournament selection process is repeated until *20* chromosomes from the population are selected to be parents.

Children are then created from the parents using crossover and mutation operators. The crossover operator uses a pair of parents and bisects the parents into two parts. Two new children are then created by combining the first part of the first parent with the second part of the second parent, and the second part of the first parent with the first part of the second parent (Figure 5.4). This creates children that have some genes from one parent and the remaining genes from the other parents. The mutation operator is used to modify the individual genes in a chromosome (Figure 5.5). The mutation operator slightly increases or decreases a gene value by a random amount.

A summary of the performance using the weightings found using the genetic algorithm can be seen in Table 5.6. The number of times each feature had a non-zero weight, with any weight $> 0.01$ considered non-zero, is presented in Table 5.7. Values of the global f-measure marked with (*) denote a statistically significant increase over the global f-measure results when using all features (p=0.01). Compared to the binary feature selection algorithm there is no increase, or a very minor increase, in the f-measure results when using the genetic algorithm.

A summary of the average weights generated using the genetic algorithm is presented in Table 5.8. The weights, for each experiment, are first normalized by dividing by the largest weight, so the largest weight becomes 1.0. It should be noted that although this table presents the average value for each weight, this average weighting is not guaranteed to provide optimal results when used in practise and may actually perform poorly.

**Table 5.6:** Performance using weights from genetic algorithm.

| | **Cases** | $f1_{global}$ | **Accuracy** | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|---|
| **Sprinter** | 5271 | 0.90(*) +/- 0.004 | 0.96 | — | 0.97 | 0.83 |
| **Tracker** | 4409 | 0.97(*) +/- 0.002 | 0.97 | — | 0.97 | 0.97 |
| **Krislet** | 4612 | 0.83(*) +/- 0.005 | 0.98 | 0.52 | 0.98 | 0.97 |
| **NoSwarm** | 4727 | 0.76(*) +/- 0.004 | 0.95 | 0.39 | 0.95 | 0.95 |
| **CMUnited** | 5078 | 0.60(*) +/- 0.002 | 0.64 | 0.55 | 0.68 | 0.59 |

### 5.2.4 Results with Dynamic Training Set

Both the backward sequential search (BSS) and genetic algorithm (GA) feature selection were run using the dynamic training set technique. The performance results

**Table 5.7:** Number of times a feature had a non-zero weight with genetic algorithm.

| | **Ball** | **Goal** | **Line** | **Flag** | **Teammate** | **Opponent** | **Unknown Player** |
|---|---|---|---|---|---|---|---|
| **Sprinter** | 2 | 25 | 4 | 7 | 1 | 3 | 1 |
| **Tracker** | 25 | 3 | 4 | 3 | 8 | 4 | 2 |
| **Krislet** | 25 | 6 | 2 | 6 | 13 | 7 | 3 |
| **NoSwarm** | 25 | 7 | 2 | 3 | 15 | 5 | 1 |
| **CMUnited** | 22 | 9 | 4 | 20 | 4 | 3 | 13 |

**Table 5.8:** Average value of feature weights with genetic algorithm.

| | **Ball** | **Goal** | **Line** | **Flag** | **Teammate** | **Opponent** | **Unknown Player** |
|---|---|---|---|---|---|---|---|
| **Sprinter** | 0.01 | 0.96 | 0.02 | 0.04 | 0.00 | 0.00 | 0.00 |
| **Tracker** | 0.98 | 0.02 | 0.02 | 0.01 | 0.06 | 0.01 | 0.00 |
| **Krislet** | 0.97 | 0.03 | 0.00 | 0.02 | 0.08 | 0.04 | 0.01 |
| **NoSwarm** | 0.98 | 0.05 | 0.00 | 0.01 | 0.11 | 0.02 | 0.00 |
| **CMUnited** | 0.85 | 0.06 | 0.01 | 0.58 | 0.02 | 0.00 | 0.09 |

can be seen for BSS is Table 5.9 and GA in Table 5.10. A summary of the number of times each feature had a non-zero weight can be seen in Table 5.11 for BSS and Table 5.12 for GA. Also, the average weights when using the GA are presented in Table 5.13. Values of the global f-measure marked with (*) denote a statistically significant increase over the global f-measure results when not using dynamic training sets (p=0.01). As with the fixed-sized training set results shown previously, we do not notice a statistically significant difference between the results from the BSS and GA algorithms.

**Table 5.9:** Performance using weights from backward sequential search using a dynamic training set.

| | $f1_{global}$ | **Accuracy** | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Sprinter** | 0.92(*) +/- 0.004 | 0.96 | — | 0.95 | 0.89 |
| **Tracker** | 0.98 +/- 0.003 | 0.98 | — | 0.98 | 0.98 |
| **Krislet** | 0.87(*) +/- 0.006 | 0.91 | 0.70 | 0.95 | 0.96 |
| **NoSwarm** | 0.85(*) +/- 0.005 | 0.90 | 0.72 | 0.91 | 0.92 |
| **CMUnited** | 0.69(*) +/- 0.004 | 0.73 | 0.69 | 0.75 | 0.63 |

**Table 5.10:** Performance using weights from genetic algorithm using a dynamic training set.

| | $f1_{global}$ | **Accuracy** | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Sprinter** | 0.92(*) +/- 0.005 | 0.97 | — | 0.94 | 0.89 |
| **Tracker** | 0.98 +/- 0.006 | 0.98 | — | 0.99 | 0.97 |
| **Krislet** | 0.86(*) +/- 0.004 | 0.90 | 0.67 | 0.93 | 0.98 |
| **NoSwarm** | 0.85(*) +/- 0.005 | 0.93 | 0.70 | 0.94 | 0.91 |
| **CMUnited** | 0.70(*) +/- 0.008 | 0.70 | 0.71 | 0.72 | 0.67 |

### 5.2.5 Analysis and Conclusions

The results show that applying feature selection, using either a binary or continuous feature selection algorithm, can significantly improve the f-measure metric. These results, however, were expected as both manual feature selection [32] and automated feature selection [20, 21] have been found to improve the imitative ability of a case-based imitation agent. It is noteworthy that feature selection has been shown to not only be useful in agents who actively play soccer (Krislet, NoSwarm and CMUnited) but also in agents with non-traditional soccer behaviour (Sprinter and Tracker).

The more interesting results arise from using the dynamic training set technique

**Table 5.11:** Number of times a feature had a non-zero weight with backward sequential search using a dynamic training set.

| | **Ball** | **Goal** | **Line** | **Flag** | **Teammate** | **Opponent** | **Unknown Player** |
|---|---|---|---|---|---|---|---|
| **Sprinter** | 0 | 25 | 0 | 0 | 0 | 0 | 0 |
| **Tracker** | 25 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Krislet** | 25 | 0 | 0 | 0 | 0 | 0 | 0 |
| **NoSwarm** | 25 | 0 | 0 | 0 | 0 | 0 | 0 |
| **CMUnited** | 25 | 0 | 0 | 25 | 0 | 0 | 0 |

described in this thesis. For the agents with fairly simple behaviour, the binary feature selection algorithm was able to find the same weights in all experiments for that agent. However, the CMUnited team found different weights in many of the binary feature selection experiments. Also, all five teams found different weightings for the different continuous feature selection experiments. However, when the dynamic training set technique is employed the weightings become the same for all experiments. Features that only had a small weight in the continuous weightings were removed in favour of the increase in case base size that their removal allowed for. The dynamic training set technique helped to prevent overfitting to the training data that occurred in the continuous weighting and to a lesser extent in the binary weighting (for the CMUnited experiments). This experiments showed that in many situations it is preferable to remove a feature if the removal of that feature will allow a larger case base to be used.

One other area of note is related to the binary feature weights found for the Krislet agent. In previous research [20], using teams of five players, the binary feature selection algorithm found both the *ball* and *teammates* to have non-zero weights using a fixed-size training case base. The reasoning for this was because the teammates

**Table 5.12:** Number of times a feature had a non-zero weight with genetic algorithm using a dynamic training set.

| | **Ball** | **Goal** | **Line** | **Flag** | **Teammate** | **Opponent** | **Unknown Player** |
|---|---|---|---|---|---|---|---|
| **Sprinter** | 0 | 25 | 0 | 0 | 0 | 0 | 0 |
| **Tracker** | 25 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Krislet** | 25 | 0 | 0 | 0 | 0 | 0 | 0 |
| **NoSwarm** | 25 | 0 | 0 | 0 | 0 | 0 | 0 |
| **CMUnited** | 25 | 0 | 0 | 25 | 0 | 0 | 0 |

tended to be located near the ball, helping the agent track the ball. However, in this work we see that only the ball is found when performing binary feature selection. This is because when a large team is used, in this case eleven players per team, the teammates occasionally block the ball from the agent's vision. This can be seen by watching the original Krislet team play. Often when there are many other players between it and the ball it begins to turn and look for the ball. When this behaviour is logged and used to create cases, two interesting cases will be create: one before the ball is blocked and one after it is blocked. These cases occur in sequence, so most of the objects will be in similar positions in both cases however the ball will only be visible in one of them. This leads to cases with highly similar teammate features that have different actions, some of which lead to the Krislet agent dashing toward the ball (when the ball is visible) and some lead to the agent turning (when the ball is temporarily blocked by the teammates). However, in both the previous research[7] and this research, the same weights were found using the dynamic-sized training case base method.

[7]The full results of this work are not presented here as a number of the experimental parameters differ (number of players, teams used, software used, etc.).

**Table 5.13:** Average value of feature weights with genetic algorithm using dynamic training set.

| | **Ball** | **Goal** | **Line** | **Flag** | **Teammate** | **Opponent** | **Unknown Player** |
|---|---|---|---|---|---|---|---|
| **Sprinter** | 0.00 | 1.0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **Tracker** | 1.0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **Krislet** | 1.0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **NoSwarm** | 1.0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **CMUnited** | 0.94 | 0.00 | 0.00 | 0.47 | 0.00 | 0.00 | 0.00 |

When examining the imitating agents playing in a game of soccer, they generally appear to behave in a manner similar to the original agents. The one exception is CMUnited. Using the techniques described in this chapter the CMUnited imitator shows some signs of the expected behaviour, like moving toward the ball, but is still has noticeable differences. Using no preprocessing, the CMUnited agent often just spins in circles or moves randomly, so the preprocessing does have a positive impact. Also, the Krislet agent occasionally exhibits some imperfect behaviour related to kicking. Since the original Krislet agent kicks toward the goal and the imitator does not use the goal (since it has a zero weight), the imitator occasionally kicks the ball in the wrong direction (toward their own goal). This is a situation where an object, the goal, is only important to one action, kicking, and not the others. Since using the goal object for reasoning may often be detrimental to dashing or turning, the goal object is removed. This is certainly a limitation as the imitating agent should never kick the ball toward its own goal.

# Chapter 6

# Case Clustering

One of the preprocessing methods we wish to examine is case prototyping. Prototyping involves replacing a set of cases with a single case, a prototypical case, that is representative of the entire set. In order for such prototyping to occur, the case base must first be divided into a number of smaller groups. The separation of data, in our situation cases, into groupings is known as data clustering [24]. Each of these clusters should ideally contain similar cases so that the prototyping process can successfully produce a case that represents the entire cluster. Ideally, the cases within a specific cluster will be nearly identical to each other so that the prototypical case is highly similar to all cases in the grouping. However, if the cases in a grouping are highly dissimilar then the prototypical case will be a less precise representation of the cluster.

By producing higher quality clusters, we would expect to produce higher quality prototypical cases. It can be seen that the quality of clustering will ultimately have an impact on the results obtained in subsequent chapters. If care is not taken to select appropriate clustering algorithms then the results in Chapters 7 and 8 may suffer.

## 6.1 Properties of Case Data

As was mentioned in previous chapters, the case data used by an imitative agent is represented by two components: the sensory inputs of the agent and the action performed. In a classification context, we can think of the sensory inputs as the *features of the data* and the action performed as the *class of the data*. This leads us to two very important properties of the data representation that we must consider when selecting an appropriate clustering algorithm:

- Variable number of known values for features
- Non-metric distance between cases

### 6.1.1 Variable Number of Known Values for Features

A case, as used by an imitative agent, contains data that represents what the agent can currently see, the state of the environment, as well as an associated action. In many domains, such as simulated RoboCup soccer, the agent has an incomplete view of the world. Since the agents can only see objects within their field of vision they can only see a subset of the objects in the world at any given time. As objects move in and out of the agent's field of vision, the number of features with known locations will change. Take, for example, Figure 6.1 representing the field of vision of a soccer playing agent at two points in time. At time $t$ the agent has a ball and a player in its field of vision, so the case is composed of two known features. At some time in the future, time *t+1*, the player moves out of the agent's field of vision leaving only the ball in the field of vision. The case representing time *t+1* would therefore only have one known feature.

In a game like soccer, we could get around this varying known feature issue by using our domain knowledge of the number of objects present in a game of soccer.

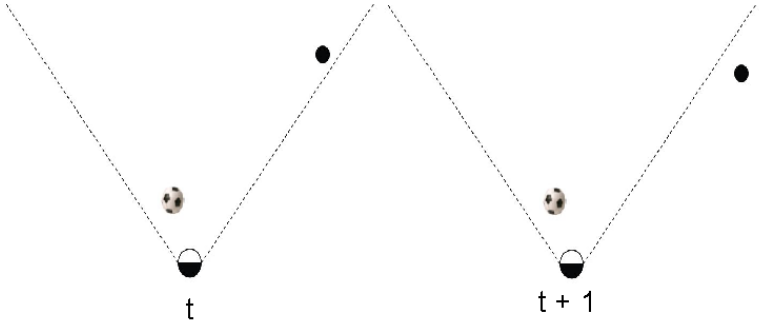


**Figure 6.1:** A change in the number of objects visible over a period of time.

We know there are a set number of players on each team, a single ball, and a fixed number of field markers (goal nets, lines or flags). This would give us a fixed sized feature space, with any objects outside the field of vision being represented as an "unknown" value. Although such a representation might be feasible in a domain like soccer, in general it is not practical because we do not know the number of objects that exist in our world. Take a human for example. Even if we limited the objects we worry about to other humans, we would still be unable to know precisely how many other humans exist in our world. Even if we knew exactly how many people lived in our city, we would likely only see a small percentage of those people at a given time. This would result in cases that are full of "unknown" values, drastically reducing the usefulness of the cases. Given the assumption of an incomplete view of the world, and therefore not knowing where every object in the world is located, we choose to compose a case using a simplified representation with only objects that are currently visible to the agent. This not only removes features with unknown values from cases, but it also removes the need for domain knowledge related to how many instances of each object exist in the world.

Thus, the world can be decomposed into a number of *feature types* representing the types of objects the agent would encounter. For example, in the game of soccer the objects could be a ball, player, flag, goal net, line or flag. Each of these feature

types would then be a multi-valued feature containing the specific instances of each object of that type. Depending on what is currently visible to the agent, each feature type could contain zero, one or many values. For example, if the agent could not currently see the soccer ball but could see three flags, the ball feature type would contain zero values whereas the flag feature type would contain three values. As the environment changes, the number of values of each feature type can change depending on what objects are currently visible to the agent.

The varying number of known features also leads to another interesting issue, in that the data can not be easily averaged. If we had a set number of features in both cases, we could represent the data as a feature vector and perform some method of averaging the feature values. In essence, the process of prototyping can be thought of as a form of averaging, since we are looking for a prototypical case that best represents an entire cluster of cases. We will see in Chapter 7 that the process of prototyping is a non-trivial problem given our data representation of cases, and as such might not be suitable to use repeatedly in a clustering algorithm. Each prototype may end up having slight *imperfections* due to the unequal number of known features in the cases used to create the prototype. These imperfections may not be significant when creating a prototypical case, but using the prototypes as "average" values in clustering algorithms may cause the algorithms to behave unexpectedly. The lack of a quality averaging method will be the most discriminating factor when selecting an appropriate clustering algorithm.

### 6.1.2 Non-metric Distance Between Cases

Many clustering algorithms work on the assumption that the distance measure, used to calculate the distance between two data points, follows the triangle inequality [60]. The triangle inequality can be conceptualized as points on a 2D plane (Figure 6.2). If there exist three points $A$, $B$ and $C$ and two of the distances between the points,

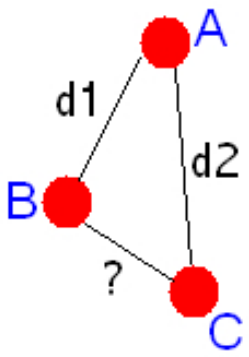


**Figure 6.2:** Relation between the distances of points.

*d1* and *d2*, are known then some sort of bounds can be placed on the possible values for the unknown distance. More specifically, the unknown value must be less than or equal to the sum of the two known distances ($d3 \leq d1 + d2$). If the triangle inequality does not hold, then the distance measurement is said to be *non-metric*[1].

### Measuring Distance Between Cases

Since the idea of clustering involves grouping *similar cases* together, it becomes necessary to define how this similarity will be judged. As we described in Chapter 4, in case-based imitation it is actually not a similarity measure that is used but rather a *dissimilarity* measure. The dissimilarity between cases is determined by calculating the distance between two cases, with low distances implying the cases are similar and large distances implying the cases are dissimilar. The following steps, as previously presented in Chapter 4, are taken when calculating the distance between two cases, $C_1$ and $C_2$:

1. **Matching:** Objects in $C_1$ are matched with corresponding objects in $C_2$. The issue of matching only really becomes important when feature types are multi-valued. For example, in soccer an agent will often see more than one *teammate*

[1]Satisfying the triangle inequality is not the only requirement for a distance measure to be metric, it must also be symmetric, positive and reflexive [60].

in its field of vision. Each of the teammates in $C_1$ must then be matched to an appropriate teammate in $C_2$. If there are an uneven number of features of a particular type between cases, the extra objects are left *unmatched*.

2. **Object-pair Distance:** The distance between each pair of matched objects is then calculated. Given that the objects represent the field of vision of an agent, each object has an associated location in the field of vision. The distance between the locations, relative to the agent, of the objects is used to calculate their object-pair distance. Any objects that went unmatched have a penalty distance applied to them.

3. **Weighting:** The object-pair distances and penalties can then have a weight applied to them. Each type of object can have a unique weight, so some objects can be given more, or less, importance. Chapter 5 focused on determining the optimal weights to use and provides more detail on feature weighting.

4. **Summation:** The weighted object-pair distances and penalties are then summed together and normalized by dividing by the total number of object-pair distances and penalties. This gives the distance between the two cases.

### Analysis of Distance Calculation

There are two main reasons why the distance calculation does not satisfy the triangle inequality, and therefore is a non-metric distance calculation. Firstly, we come back to the issue of multi-valued feature types, depending on how many objects were in the agent's field of vision. If we had a complete world view, and therefore a fixed number of features, the data could be represented as a feature vector and a metric distance calculation, such as Euclidean distance, could be used. This becomes an issue when the notion of *penalties* comes in.

For example, take Figure 6.3. All three cases have a different set of known features. Case $A$ has a ball, case $B$ has a ball and player and case $C$ has a ball and goal. When calculating the distance between $A$ and either $B$ or $C$, the distance calculation must take into account $A$ has one less feature than the others (in one case it is missing a player and in the other it is missing a goal). However when $B$ and $C$ have their distance computed, there are two features that do not match because $B$ does not have a goal and $C$ does not have a player. Unless the distance calculation completely ignores features that are only present in one case, then this type of situation can lead to a non-metric distance. In our example, *distance(A,B)* and *distance(A,C)* would have encountered one case of feature mismatch (and an associated penalty added to their distance) whereas *distance(B,C)* would encounter two such penalties. So although the distance measuring how close the ball objects are in each case is metric, the penalties added would make the distance non-metric. For example, assume the ball is at the same position in each case and therefore does not impact the distance between cases. We get the following distances composed entirely of penalties, $P_i$:

$$\begin{aligned} distance(A,B) &= P_{player} \\ distance(A,C) &= P_{goal} \\ distance(B,C) &= \frac{P_{player} + P_{goal}}{2} \end{aligned}$$

For the triangle inequality to hold:

$$\begin{aligned} distance(A,B) &\leq distance(A,C) + distance(B,C) \\ P_{player} &\leq P_{goal} + \frac{P_{player} + P_{goal}}{2} \end{aligned}$$

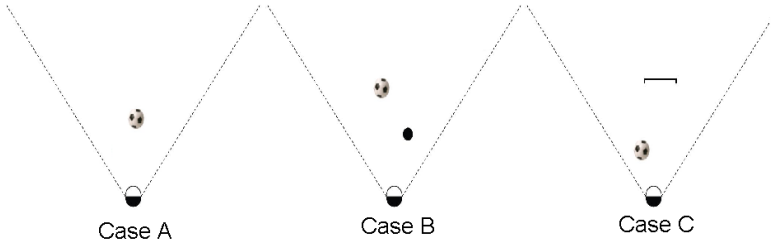


**Figure 6.3:** Three cases with different sets of feature types.

But if $P_{goal} = 0$, we get $P_{player} \leq \frac{P_{player}}{2}$ which is not possible unless $P_{player} = 0$. While this shows a situation where the triangle inequality *does not* hold, in most situations it will hold. However, given that there are certain situations where the triangle inequality does not hold, the distance calculation can not be said to be metric.

Secondly, we have the issue of object matching. In the previous example we had a situation where each case had different sets of objects in them. For this situation, we will consider when each case has multiple instances of the same object. For example, in Figure 6.4, each case has two players in it. As described in Chapter 4, when calculating the distance between a pair of cases, we must first match each object in one case with a corresponding object from the second case (Algorithm 1 in Chapter 4). A possible matching is given in Figure 6.4, and if we examine the way the objects are matched we can see where the triangle inequality might be broken. When case *1* is matched with cases *2* and *3*, object $a$ is matched with objects $c$ and $e$ respectively. However, when we match cases *2* and *3* objects $c$ and $e$ do not match with each other. Ideally, the objects would all match with each other, similar to Figure 6.2 that we discussed previously. However in this case, the unknown distance in Figure 6.2 would not be the distance between $B$ and $C$, but rather the distance between $B$ and a fourth point (or between $C$ and another point). This would not produce a triangular relation between the three points since $B$ and $C$ would not be connected. One of the key parts of the distance calculation, matching, causes a non-metric distance calculation and may cause many clustering algorithms to behave differently than expected since many

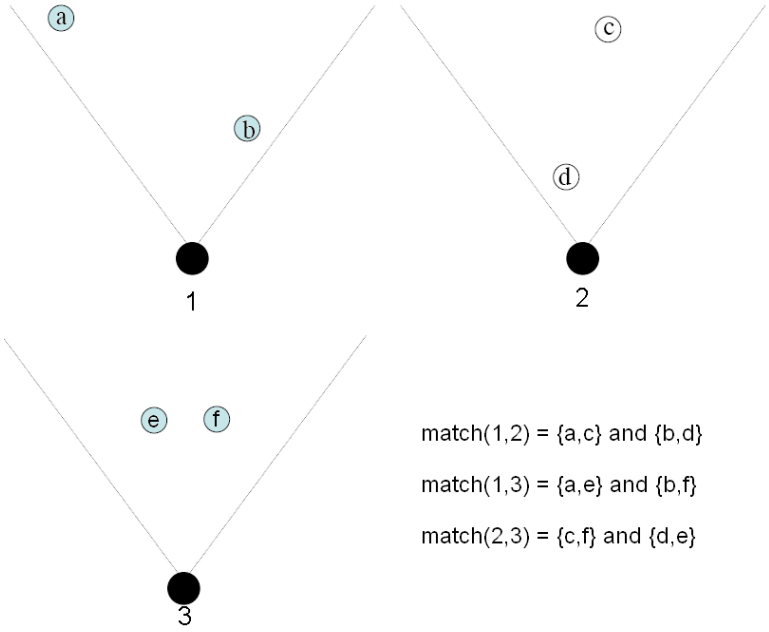


**Figure 6.4:** Object matching between three cases.

make an assumption of having a metric distance calculation [60].

### 6.1.3 Limitation of Algorithms

Based on the properties of the data representation and distance measurement that were mentioned above, a large number of algorithms can be excluded from possible analysis because they are not appropriate for our needs.

- Algorithms that rely on having access to specific features values, such as the minimum mutation algorithm [24] or spiral search algorithm [24], are not suitable because there is no guarantee that any one particular feature will have a known value in all cases (or even a majority of the cases).

- Any algorithm that involves some "average" cluster value to act as a cluster centre. This includes any algorithm that uses a centroid value to calculate within-cluster error or between-cluster error. This eliminates algorithms such

as k-means (and many variants of it) [39], mixtures algorithm [24], fisher algorithm [24] and the binary splitting algorithm [24]. In a similar way, the adding and joining algorithms [24] are also inappropriate because they require the merging of data points together. Such a merger would be similar to creating a prototypical case.

## 6.2 Analysis of Clustering Algorithms

In previous sections of this chapter, criteria were presented for discriminating against certain types of clustering algorithms based on the properties of the data we are trying to cluster. This section, on the other hand, will present several types of clustering methods that can potentially be applied to imitative case data. It should be noted that it is outside the scope of this analysis to utilize all potential clustering algorithms. Instead, this analysis will focus on several different types of clustering algorithms and compare the results obtained with each.

### 6.2.1 Leader Algorithm

The first clustering method we will consider is Hartigan's leader algorithm [24]. The algorithm works with a set, $N$, of data points and partitions the data into a number of clusters, given a threshold value $T$. The algorithm is described in Algorithm 6.

As we can see, this algorithm requires only a single pass through the data so it can execute quite quickly. However, using only a single pass can also be a limitation because the algorithm will be unlikely to cluster as accurately as a multi-pass algorithm.

Another downfall of the algorithm is that it is highly sensitive to the ordering of the data. The algorithm locates the first cluster leader that is within the threshold, not necessarily the closest cluster leader. This can result in the first clusters being

**Algorithm 6** Leader Algorithm

**Inputs:** N, T
**Outputs:** allClusters: the clusters created by the algorithm

```
select the first case, C, from N
place C in a new cluster, G
add G to allClusters
set C as the leader of G
while(more cases remain in N):
   select the next case, C
   while(more cluster leaders remain)
      select the next cluster leader, L
      if(distance(C,L) < T)
         place C in the same cluster as L
         stop loop
   end loop
   if(C was not placed in a cluster)
      create new cluster G
      add C to G and make C the leader
      add G to allClusters
end loop
return allClusters
```

larger than later clusters. As well, if the order of the data is changed, the resulting clustering will change.

We should also note that the cases in a cluster will be within a threshold value of the cluster leader, but we can make no guarantees as to how close they will be to each other. This is a result of the violation of the triangle inequality, as we described earlier. Even if the threshold value used was quite small, meaning all cases are a small distance from the leader, the distances between the non-leader cases could be quite large. It this situation, we might expect this type of clustering to work best when the leader of the cluster is used as a prototype for the rest of the cluster, as opposed to creating a prototype that represents an average of the cases in the cluster.

Although numerous limitations of this algorithm have been discussed, the leader algorithm will be used for analysis, in part, because of these limitations. As a simple, single pass algorithm the leader algorithm can be used as a benchmark for the more

complex and computationally intensive algorithms.

### 6.2.2 Agglomerative Clustering

The central piece of data used by agglomerative clustering algorithms is the distance matrix. Given an ordered set of $N$ items, the distance matrix, $M$, will be an $NxN$ matrix with the value at *M(i,j)* being the distance between items $i$ and $j$.

In the single-linkage algorithm [24] the set of cases, $N$, along with a threshold value, $T$, is given as input to the algorithm, which is described in Algorithm 7.

**Algorithm 7** Single-linkage Algorithm

**Inputs:** N, T

```
for(int ii=0; ii < N.length; ii++)
   C1 = N[ii]
   for(int jj=0; jj < N.length; jj++)
      C2 = N[jj]
      distance = distance(C1,C2)
      M[ii,jj] = distance
   end loop
end loop
while(a value in matrix M < T):
     select the minimum value in matrix M, at point M(i,j)
     combine the items i and j into a cluster
     combine rows i and j using the smaller of the two values for each entry
     combine columns i and j using the smaller of the two values for each entry
end loop
```

This algorithm maintains a matrix, the distance matrix, that holds information on the closest link (shortest path) between each cluster. Initially, each cluster is composed of a single data item but can grow over time. The algorithm will iteratively combine clusters until no two clusters are closer than the threshold distance. It should also be noted that as clusters are formed, the size of the matrix will decrease, with pairs of rows and columns being replaced by a single row and column.

Unlike the leader algorithm, this algorithm is not reliant on the ordering of the data

because all distances are calculated before the clustering process begins. However, similar to the leader algorithm we have no guarantee that all items in a cluster will be close to each other. This is because the distances used are the minimum distance a cluster is from another cluster, so although one item in a cluster may be close to another data point, the other items in the cluster may be quite far away.

Although the single-linkage algorithm was selected to be used for analysis, a number of other agglomerative algorithms could also potential be used as well. The average-linkage algorithm does not take the minimum values when combining rows and columns in the distance matrix, but instead uses a mean value. The complete-linkage algorithm [30] can be thought of as the opposite of single-linkage, because the maximum distance to each other cluster is used when combining rows and columns.

### 6.2.3 Distance Feature Vector

The distance feature vector approach can be thought of as a method for mapping a data set into a feature vector representation. As we mentioned earlier, the imitative case data only contains information regarding objects it can currently see, so each case can contain a different number of known features and can not be directly represented in a feature vector form. This approach, however, uses the distances between cases as features.

In the basic form, consider an ordered data set with $N$ items. For each of the $N$ items, a feature vector of size $N$ will be created. The feature vector for a specific item, $I$, will contain the distance between $I$ and all items in the data set (including itself). The feature vectors can then be clustered using a traditional clustering method. It should be noted that the feature vector can be smaller than size $N$ if only a subset of the items are used to produce the feature vector (if a set of representative items are known, for example). One study found this approach to outperform other clustering methods on data with a non-metric distance calculation when tested experimentally

[10].

For our analysis, we will use the entire data set to create the feature vector ($N$ features) and the k-means algorithm [39] to perform clustering. In k-means clustering, a fixed number of clusters are produced, as given by a parameter of the algorithm. Each cluster has a centroid value which stores the average value of the data points in that cluster. Initially, the cluster centroids are generated randomly and each data item is assigned to the cluster with the centroid that is closest to itself. For each cluster, the centroid is then recalculated by averaging all data points that are members of the cluster. The process of reassigning data points and recalculating cluster centroids continues for a fixed number of iterations or until no changes occur in the clusters.

We would expect this method to produce clusters which best overcome the issue of the triangle inequality. Since each feature in the distance feature vector represents the distance to a specific data point, points that are clustered together will have similar distance vectors and will therefore have similar distances to the data points in the data set.

## 6.3 Cluster Evaluation

The goal of clustering is to partition data into a number of groups, with each group being composed a subset of the original data points. Each clustering algorithm performs the partitioning in a slightly different manner which will likely lead to different groupings of the data. Not only can different algorithms partition the data differently, but an algorithm can also partition the data differently if the algorithm parameters are changed. As well, an algorithm may produce different partitions depending on the order of the data or if any random selection is used.

Given that the data can be partitioned into a variety of different clusters, it then becomes necessary to examine the quality of a particular clustering. If we make no

assumptions about the nature of the data or how we *assume* the data should be clustered, the validity of a cluster can be measured by examining two properties of the clusters: *consistency* [55] and *compactness and separation* [9].

### 6.3.1 Consistency

The consistency of a clustering relates to how "*clean*" each of the individual clusters are. The case-based reasoning data that we utilize is, by its very nature, labelled if we consider the action of a case to be the class the case belongs to. Given that we know the class of each case, we ideally want a clustering that puts items of the same class into the same cluster[2]. We will measure the consistency using the purity [55] and entropy measures [62].

#### Purity

The purity measure represents the number of data items that are in a cluster where their class is the dominant class. This makes the assumption that the majority class in a cluster is the correct class for that cluster. The following notation is used for both the purity and entropy equations:

$nc$: The number of clusters ( $C_i \in \{C_1, ..., C_{nc}\}$ )

$na$: The number of classes ( $A_i \in \{A_1, ..., A_{na}\}$ )

$n$: The total number of data items

$n_i$: The number of data items in cluster $i$

$n_i^j$: The number of data items in cluster $i$ that belong to class $j$

The percentage of cluster $i$ that is of class $j$ is denoted by $p_i^j$:

[2]It should be noted that optimal consistency values may not always be obtainable, particularly in cases with significant amounts of noise or with missing features. Both of these situations could result in identical data with different classes.

$$p_i^j = \frac{n_i^j}{n_i}$$

We can then calculate the purity of a clustering using Equation 6.1.

$$Purity = \frac{1}{n} \sum_{i=1}^{nc} \max_{j=1}^{na} p_i^j \tag{6.1}$$

It should be noted that during experimentation we only include clusters with multiple data points in the purity calculation. This is to promote clustering multiple items in order to help reduce the case base size. For example, a clustering could obtain a high purity score by having all clusters, except one, contain only one data point. The only cluster that could be impure would be the one with multiple data points, so the overall purity would be a large value, since nearly all clusters are pure, without having performed any useful clustering.

The purity value will be in the range [0,1], with a higher value representing a more pure clustering. Therefore, we want to maximize the purity value. We can see that the optimum value will occur when a value of $p_i^j = 1$ is attained for every cluster (each cluster is only composed on a single class).

**Entropy**

The entropy measures the disorder of the clustering. As with the purity, we ideally want the clusters to only contain data items of the same class. Using the same notation as we used with the purity equation we define the entropy of a single cluster as:

$$E(C_i) = -\sum_{j=1}^{na} p_i^j \log p_i^j \tag{6.2}$$

We can then calculate the entropy of the entire clustering as:

$$E = \frac{1}{n} \sum_{i=1}^{nc} n_i E(C_i) \tag{6.3}$$

The optimum entropy value occurs when the entropy of each cluster, $E(C_i)$, is zero. This makes the overall entropy zero and which occurs when a value of $p_i^j = 1$ exists in every cluster (which will make all other $p_i^j$ values in the cluster zero). As such, the entropy value should be minimized. As with the purity value, during experimentation only clusters with multiple data points are included in the entropy measurement.

While purity and entropy are similar measures, both were used because entropy takes into account the percentage of a cluster that is of *each* action type, not just the dominant action type. This is because purity only uses the largest $p_i^j$ value, whereas entropy uses the $p_i^j$ values for all actions. Thus, both metrics were included in order to see if they may lead to different conclusions about the algorithms being tested.

### 6.3.2 Compactness and Separation

Two important properties of a clustering, that consider the location and distribution of clusters, are the compactness and separation. The *compactness* refers to how close the data items in a cluster are. Ideally we would like to have all data items very close to each other. In order to measure the compactness we use the intra-cluster distance. We will use the maximum distance between any two items in a cluster as the intra-cluster distance:

$$\delta(C_i) = \max_{x,y \in C_i} distance(x, y) \tag{6.4}$$

$C_i$: The *ith* cluster

$x, y$: Data items in cluster $C_i$

$distance(x, y)$: A measure of distance between two data items

We could have used another measure of intra-cluster distance (minimum distance, average distance, or some other measure) however we felt using the maximum distance between items in the cluster provided the most conservative measure of intra-cluster distance. A low intra-cluster distance implies that the data items in the cluster are close to each other.

The *separation* of clusters measures inter-cluster distance (how close the clusters are to each other). This measure is important because it can help identify algorithms that over-partition data. For example, breaking a large cluster into two smaller clusters will likely result in a net decrease in the inter-cluster distance because the two clusters will usually be quite close to each other. As with the intra-cluster distance there are a number of possible ways to calculate the inter-cluster distance (distance between cluster centroids or distance between furthest points of the clusters are two other possibilities) however we will use the single-linkage distance. The single-linkage distance is the closest distance between two clusters and is measured as:

$$\Delta(C_i, C_j) = \min_{x \in C_i, y \in C_j} distance(x, y) \tag{6.5}$$

Now that we have established measurements for the compactness and separation of clusters, we can then combine these values into measures that express the compactness and separation of the entire clustering. In order to do this we will use the Davies-Bouldin Index [15] and Dunn Index [18].

Unlike with the purity and entropy measures, clusters that only contain a single data point will be included when calculating the Davies-Bouldin Index and Dunn Index. This is because both of these measures include terms, the inter-cluster distance, that will penalize under-partitioning of the data.

### 6.3.3 Davies-Bouldin Index

The Davies-Bouldin Index attempts to maximize all inter-cluster distance (between every possible pair of clusters) while at the same time minimizing intra-cluster distance for each cluster. The equation for the Davies-Bouldin Index is as follows:

$$DB = \frac{1}{nc} \sum_{i=1}^{nc} \max_{j=1, j \neq i}^{nc} \frac{\delta(C_i) + \delta(C_j)}{\Delta(C_i, C_j)} \tag{6.6}$$

$nc$: The number of clusters

We can see from the equation that large inter-cluster distances and small intra-cluster distances will result in low values of the Davies-Bouldin Index, therefore we try and minimize this value.

### 6.3.4 Dunn Index

As was the case with the Davies-Bouldin Index, the Dunn Index attempts to measure the compactness and separation of a clustering. We calculate it as follows:

$$Dunn = \min_{i=1}^{nc} \left\{ \min_{j=1, j \neq i}^{nc} \frac{\Delta(C_i, C_j)}{\max_{k=1}^{nc} \delta(C_k)} \right\} \tag{6.7}$$

The Dunn Index will effectively be the smallest inter-cluster distance divided by the largest intra-cluster distance. This will provide a worst-case scenario by combining the worst separability with the worst compactness. By maximizing this value we help to reduce the worst cases of separability and compactness in our clustering.

## 6.4 Experimental Results

For the experiments in this chapter we wish to examine the various clustering algorithms and attempt to determine if any of the algorithms perform significantly better

than the others. The goal of this chapter is to select one of the algorithms so that it can be used in subsequent chapters to assist with the prototyping process (Chapter 7). For the various algorithms, the purity and Dunn Index should be *maximized* whereas the entropy and Davies-Bouldin Index should be *minimized*.

In order to accurately compare the various algorithms, each algorithm will be used to prototype a case base containing a multiple of *6000* cases. The sizes of case bases used are 6000, 12000, 18000 and 24000 cases. These sizes for case bases were chosen because they represent multiples of the maximum number of cases that can be created by watching a single game of simulated RoboCup soccer, since each game each player gets 6000 sensory inputs and can perform 6000 actions in a game. Therefore, the sizes represent the data generated by watching one, two, three and four complete games. For each size of case base, 25 sets of cases were generated for each type of agent by selecting the cases randomly from a larger case set. The results presented for each algorithm will be the mean values for each metric, since each algorithm will cluster each of the 25 data sets.

### 6.4.1 Number of Clusters

Each of the initial case bases contain *at least* 6000 cases, but as we found in Chapter 5 the number of cases that can be searched within the real-time limit of a simulated RoboCup agent is less than 6000. Looking back at Table 5.1, we can see the maximum number of cases that can be searched within the imposed time limit for each of the agents. Since these are the case base limits, clustering will be used to compress the initial case bases to these maximum sizes.

It then becomes necessary to set the parameters of the clustering algorithms so that they will produce the number of clusters we want. When using the distance feature vector approach, the case data is represented as a fixed length feature vector that we can then apply the k-means clustering algorithm to. One of the parameters

of the k-means algorithm, the k-value, specifies how many clusters will be produced. For this algorithm, we can simply supply the number of clusters as a parameter to the k-means algorithm. However, for both the leader and single-linkage algorithm the number of clusters is not directly supplied. The parameter those algorithms do have is the *threshold* parameter that tells the algorithm the maximum distance between data points for them to still be in the same cluster. For the experiments involving the leader and single-linkage algorithm, the threshold parameter will be tested for a range of possible values in order to determine which threshold value will produce the desired number of clusters.

### 6.4.2 Results

We have divided the results, for each initial case base size, into each of the five agent types that are tested. The following tables show the results:

- **6000 initial cases:** Sprinter (Table 6.1), Tracker (Table 6.2), Krislet (Table 6.3), NoSwarm (Table 6.4) and CMUnited (Table 6.5)
- **12000 initial cases:** Sprinter (Table 6.6), Tracker (Table 6.7), Krislet (Table 6.8), NoSwarm (Table 6.9) and CMUnited (Table 6.10)
- **18000 initial cases:** Sprinter (Table 6.11), Tracker (Table 6.12), Krislet (Table 6.13), NoSwarm (Table 6.14) and CMUnited (Table 6.15)
- **24000 initial cases:** Sprinter (Table 6.16), Tracker (Table 6.17), Krislet (Table 6.18), NoSwarm (Table 6.19) and CMUnited (Table 6.20)

**Table 6.1:** Clustering Algorithms Comparison - 6000 Initial Cases - Sprinter

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.88 +/- 0.006 | 0.19 +/- 0.004 | 0.39 +/- 0.011 | 0.26 +/- 0.014 |
| **Single-linkage** | 0.93 +/- 0.004 | 0.11 +/- 0.005 | 0.25 +/- 0.015 | 0.49 +/- 0.017 |
| **Distance Vector** | 0.98 +/- 0.004 | 0.07 +/- 0.004 | 0.17 +/- 0.009 | 0.60 +/- 0.013 |

**Table 6.2:** Clustering Algorithms Comparison - 6000 Initial Cases - Tracker

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.83 +/- 0.007 | 0.27 +/- 0.005 | 0.46 +/- 0.010 | 0.22 +/- 0.016 |
| **Single-linkage** | 0.90 +/- 0.004 | 0.15 +/- 0.004 | 0.28 +/- 0.009 | 0.44 +/- 0.012 |
| **Distance Vector** | 0.93 +/- 0.005 | 0.10 +/- 0.007 | 0.25 +/- 0.007 | 0.50 +/- 0.012 |

**Table 6.3:** Clustering Algorithms Comparison - 6000 Initial Cases - Krislet

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.85 +/- 0.009 | 0.24 +/- 0.008 | 0.45 +/- 0.016 | 0.24 +/- 0.013 |
| **Single-linkage** | 0.90 +/- 0.004 | 0.15 +/- 0.006 | 0.25 +/- 0.013 | 0.46 +/- 0.014 |
| **Distance Vector** | 0.94 +/- 0.006 | 0.10 +/- 0.004 | 0.22 +/- 0.008 | 0.54 +/- 0.011 |

**Table 6.4:** Clustering Algorithms Comparison - 6000 Initial Cases - NoSwarm

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.84 +/- 0.009 | 0.25 +/- 0.007 | 0.47 +/- 0.018 | 0.23 +/- 0.010 |
| **Single-linkage** | 0.89 +/- 0.005 | 0.17 +/- 0.005 | 0.27 +/- 0.011 | 0.42 +/- 0.009 |
| **Distance Vector** | 0.94 +/- 0.007 | 0.11 +/- 0.006 | 0.23 +/- 0.010 | 0.54 +/- 0.012 |

**Table 6.5:** Clustering Algorithms Comparison - 6000 Initial Cases - CMUnited

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.80 +/- 0.008 | 0.34 +/- 0.010 | 0.59 +/- 0.014 | 0.16 +/- 0.009 |
| **Single-linkage** | 0.85 +/- 0.007 | 0.24 +/- 0.006 | 0.44 +/- 0.011 | 0.25 +/- 0.008 |
| **Distance Vector** | 0.89 +/- 0.004 | 0.16 +/- 0.009 | 0.25 +/- 0.012 | 0.43 +/- 0.014 |

**Table 6.6:** Clustering Algorithms Comparison - 12000 Initial Cases - Sprinter

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.80 +/- 0.005 | 0.23 +/- 0.007 | 0.38 +/- 0.009 | 0.25 +/- 0.013 |
| **Single-linkage** | 0.90 +/- 0.005 | 0.13 +/- 0.004 | 0.24 +/- 0.008 | 0.49 +/- 0.009 |
| **Distance Vector** | 0.96 +/- 0.007 | 0.08 +/- 0.005 | 0.14 +/- 0.012 | 0.61 +/- 0.015 |

**Table 6.7:** Clustering Algorithms Comparison - 12000 Initial Cases - Tracker

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.82 +/- 0.004 | 0.25 +/- 0.007 | 0.44 +/- 0.008 | 0.28 +/- 0.012 |
| **Single-linkage** | 0.88 +/- 0.005 | 0.15 +/- 0.003 | 0.25 +/- 0.007 | 0.46 +/- 0.010 |
| **Distance Vector** | 0.90 +/- 0.005 | 0.14 +/- 0.005 | 0.23 +/- 0.009 | 0.50 +/- 0.014 |

**Table 6.8:** Clustering Algorithms Comparison - 12000 Initial Cases - Krislet

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.83 +/- 0.007 | 0.24 +/- 0.005 | 0.43 +/- 0.016 | 0.25 +/- 0.013 |
| **Single-linkage** | 0.87 +/- 0.005 | 0.16 +/- 0.008 | 0.24 +/- 0.013 | 0.45 +/- 0.011 |
| **Distance Vector** | 0.93 +/- 0.007 | 0.10 +/- 0.005 | 0.20 +/- 0.009 | 0.54 +/- 0.014 |

**Table 6.9:** Clustering Algorithms Comparison - 12000 Initial Cases - NoSwarm

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.83 +/- 0.006 | 0.25 +/- 0.005 | 0.46 +/- 0.014 | 0.25 +/- 0.018 |
| **Single-linkage** | 0.87 +/- 0.006 | 0.19 +/- 0.003 | 0.25 +/- 0.009 | 0.43 +/- 0.012 |
| **Distance Vector** | 0.92 +/- 0.005 | 0.13 +/- 0.008 | 0.21 +/- 0.011 | 0.54 +/- 0.010 |

**Table 6.10:** Clustering Algorithms Comparison - 12000 Initial Cases - CMUnited

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.79 +/- 0.009 | 0.34 +/- 0.007 | 0.59 +/- 0.010 | 0.16 +/- 0.008 |
| **Single-linkage** | 0.84 +/- 0.008 | 0.25 +/- 0.006 | 0.42 +/- 0.014 | 0.28 +/- 0.019 |
| **Distance Vector** | 0.87 +/- 0.005 | 0.18 +/- 0.008 | 0.24 +/- 0.014 | 0.40 +/- 0.011 |

**Table 6.11:** Clustering Algorithms Comparison - 18000 Initial Cases - Sprinter

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.74 +/- 0.005 | 0.28 +/- 0.007 | 0.54 +/- 0.013 | 0.20 +/- 0.011 |
| **Single-linkage** | 0.86 +/- 0.007 | 0.24 +/- 0.006 | 0.45 +/- 0.012 | 0.27 +/- 0.011 |
| **Distance Vector** | 0.90 +/- 0.005 | 0.12 +/- 0.008 | 0.25 +/- 0.009 | 0.51 +/- 0.013 |

**Table 6.12:** Clustering Algorithms Comparison - 18000 Initial Cases - Tracker

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.77 +/- 0.004 | 0.27 +/- 0.006 | 0.47 +/- 0.014 | 0.24 +/- 0.012 |
| **Single-linkage** | 0.83 +/- 0.007 | 0.25 +/- 0.007 | 0.48 +/- 0.011 | 0.23 +/- 0.009 |
| **Distance Vector** | 0.86 +/- 0.007 | 0.23 +/- 0.008 | 0.44 +/- 0.010 | 0.29 +/- 0.012 |

### 6.4.3 Analysis and Conclusions

For all of the different agents and initial case base sizes, a clear trend emerges. Examining all four metrics, the distance feature vector approach performs significantly better than the leader or single-linkage algorithms. For the purity and Dunn Index values, the distance feature vector results are a statistically significant increase and the entropy and Davies-Bouldin Index are a statistically significant decrease over the other algorithms (with p=0.01). The only exception to this is when the initial case base is of size 24000. For that size, the distance vector approach is as good or better

**Table 6.13:** Clustering Algorithms Comparison - 18000 Initial Cases - Krislet

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.76 +/- 0.004 | 0.28 +/- 0.008 | 0.46 +/- 0.009 | 0.21 +/- 0.011 |
| **Single-linkage** | 0.83 +/- 0.006 | 0.24 +/- 0.007 | 0.45 +/- 0.009 | 0.22 +/- 0.010 |
| **Distance Vector** | 0.90 +/- 0.005 | 0.12 +/- 0.007 | 0.21 +/- 0.013 | 0.51 +/- 0.009 |

**Table 6.14:** Clustering Algorithms Comparison - 18000 Initial Cases - NoSwarm

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.75 +/- 0.008 | 0.28 +/- 0.009 | 0.46 +/- 0.013 | 0.22 +/- 0.010 |
| **Single-linkage** | 0.83 +/- 0.006 | 0.23 +/- 0.005 | 0.45 +/- 0.010 | 0.23 +/- 0.010 |
| **Distance Vector** | 0.89 +/- 0.006 | 0.12 +/- 0.005 | 0.22 +/- 0.007 | 0.50 +/- 0.009 |

**Table 6.15:** Clustering Algorithms Comparison - 18000 Initial Cases - CMUnited

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.75 +/- 0.006 | 0.36 +/- 0.004 | 0.61 +/- 0.012 | 0.15 +/- 0.011 |
| **Single-linkage** | 0.82 +/- 0.007 | 0.28 +/- 0.006 | 0.44 +/- 0.013 | 0.25 +/- 0.015 |
| **Distance Vector** | 0.83 +/- 0.005 | 0.26 +/- 0.004 | 0.44 +/- 0.007 | 0.27 +/- 0.006 |

**Table 6.16:** Clustering Algorithms Comparison - 24000 Initial Cases - Sprinter

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.64 +/- 0.010 | 0.37 +/- 0.009 | 0.63 +/- 0.014 | 0.17 +/- 0.016 |
| **Single-linkage** | 0.77 +/- 0.009 | 0.26 +/- 0.008 | 0.51 +/- 0.012 | 0.21 +/- 0.011 |
| **Distance Vector** | 0.83 +/- 0.009 | 0.25 +/- 0.006 | 0.48 +/- 0.013 | 0.23 +/- 0.013 |

**Table 6.17:** Clustering Algorithms Comparison - 24000 Initial Cases - Tracker

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.66 +/- 0.008 | 0.33 +/- 0.005 | 0.57 +/- 0.008 | 0.19 +/- 0.010 |
| **Single-linkage** | 0.78 +/- 0.009 | 0.26 +/- 0.007 | 0.50 +/- 0.010 | 0.22 +/- 0.012 |
| **Distance Vector** | 0.80 +/- 0.008 | 0.23 +/- 0.010 | 0.47 +/- 0.015 | 0.25 +/- 0.017 |

**Table 6.18:** Clustering Algorithms Comparison - 24000 Initial Cases - Krislet

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.70 +/- 0.011 | 0.29 +/- 0.013 | 0.55 +/- 0.012 | 0.18 +/- 0.012 |
| **Single-linkage** | 0.81 +/- 0.006 | 0.25 +/- 0.007 | 0.48 +/- 0.015 | 0.22 +/- 0.016 |
| **Distance Vector** | 0.85 +/- 0.009 | 0.22 +/- 0.011 | 0.45 +/- 0.012 | 0.24 +/- 0.010 |

**Table 6.19:** Clustering Algorithms Comparison - 24000 Initial Cases - NoSwarm

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.68 +/- 0.009 | 0.30 +/- 0.007 | 0.56 +/- 0.015 | 0.18 +/- 0.018 |
| **Single-linkage** | 0.82 +/- 0.012 | 0.23 +/- 0.009 | 0.46 +/- 0.012 | 0.23 +/- 0.014 |
| **Distance Vector** | 0.84 +/- 0.010 | 0.23 +/- 0.007 | 0.45 +/- 0.013 | 0.24 +/- 0.016 |

**Table 6.20:** Clustering Algorithms Comparison - 24000 Initial Cases - CMUnited

| | **Purity** | **Entropy** | **DB Index** | **Dunn Index** |
|---|---|---|---|---|
| **Leader** | 0.70 +/- 0.011 | 0.39 +/- 0.013 | 0.63 +/- 0.008 | 0.13 +/- 0.011 |
| **Single-linkage** | 0.76 +/- 0.007 | 0.36 +/- 0.009 | 0.61 +/- 0.016 | 0.13 +/- 0.014 |
| **Distance Vector** | 0.76 +/- 0.010 | 0.35 +/- 0.009 | 0.58 +/- 0.015 | 0.15 +/- 0.010 |

for all metrics but not at a statistically significant level.

Such a distinct advantage for the distance feature vector approach, using k-means clustering, makes selecting the algorithm to use for prototyping a simple choice. Also, since this approach used the k-means algorithm, which allows the number of output clusters to be set as a parameter, producing the desired number of output clusters required less work than with the leader and single-linkage algorithm. Both the single-linkage and leader algorithms require modifying their threshold parameters in order to get the desired number of output clusters, adding more work to the preprocessing processes.

Intuitively, we expected the leader algorithm to perform worse compared to the other algorithms since it simply adds a case to the *first* cluster it could belong to, and not necessarily the cluster that it is the closest to. As was mentioned when the leader algorithm was presented, the initial clusters tend to contain many cases whereas the remaining cluster often only contain a single case. This can cause the initial clusters to have a higher chance of being impure, since cases tend to be added more often to the initial clusters. Also, we noted cases are added to the first cluster they could belong to and may not be added to the best cluster for them. This can cause the inter-cluster distances to be small, leading to poor Dunn Index and Davies-Bouldin Index values.

While the single-linkage algorithm performed well compared to the leader algorithm, it performed significantly worse compared to the distance feature vector method. One of the reasons it likely performed well is because it combines clusters that are close to each other using the *minimum* distance between clusters. This behaviour helps to minimize inter-cluster distance, which is a key factor in many of the metric. However, due to the non-metric nature of the distance calculation there is no guarantee the intra-cluster distances will be small. Also, the single-linkage algorithm only performs a single clustering of the data. The distance feature vector approach

uses k-means clustering, which creates an initial clustering and then tries to improve that clustering in an iterative process. This iterative refining of the clustering is likely a primary reason the distance feature vector approach performed best.

Another area of note is that the distance feature vector approach was used to represent the data as a feature vector, with the features representing the distance to other cases. This data was then used by a k-means algorithm to perform the clustering. Although the k-means algorithm was used, the distance feature vector approach opens the door to a wide variety of clustering algorithms that were previously unsuitable for clustering the case data. An area of future work would be to use the distance feature vector method with other clustering algorithms to see if an improvement could be made over the results using the k-means algorithm.

When comparing the clustering results using different sizes for the initial case base, another trend emerges. We can see for the various agent types that the purity and entropy values decrease as the size of the initial case base is increased. This is likely due to the higher level of compression required to compress a larger case base to a case base of a fixed size. As the number of cases increases, so to does the likelihood that cases of a different class will be put in the same cluster.

However, if we examine the Davies-Bouldin Index and Dunn Index we see there is actual an increase in these values, using an initial case base of *12000* cases, before they decline for larger sized case bases. This increase can be explained by examining the number of single-case clusters. When clustering *6000* cases, there would be many single-case clusters and they could have a low inter-cluster distance between them if the cases were not uniformly distributed over the problem space. When more cases are added, a more representative sample of the problem space is obtained leading to the clusters being more likely to be spaced apart. However, after a certain point adding more cases will not have a significant impact on increasing the representation of the problem space, but will instead lead to more cases in existing areas of the problem

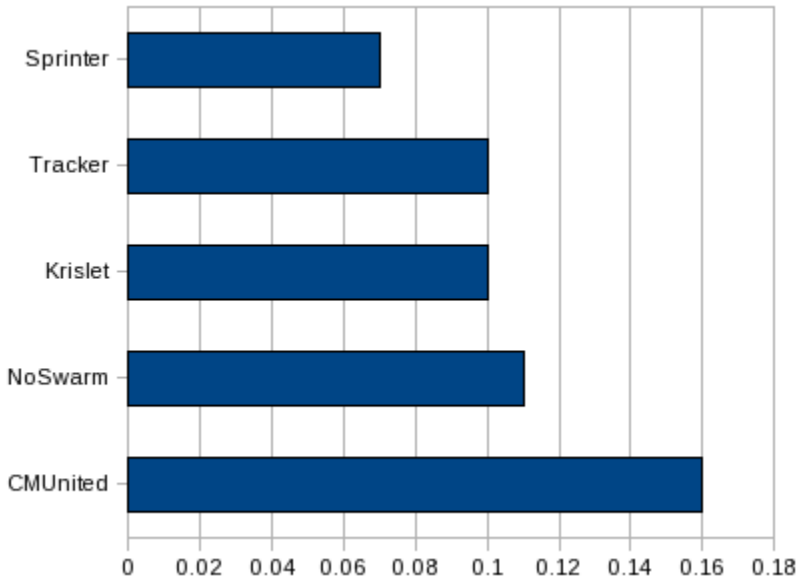


**Figure 6.5:** The entropy values for the agents using an initial case base of size 6000.

space. Having more cases in each cluster can then lead to the intra-cluster distance increasing and the inter-cluster distance decreasing, resulting in the Davies-Bouldin Index and Dunn Index moving away from their ideal values.

Lastly, the metrics can be used to gain insight into the complexity of an agent being imitated. For example, looking at Figure 6.5 we can see that CMUnited has the highest entropy value while Sprinter has the lowest. Tracker, Krislet and NoSwarm all have very similar values which we would expect since they behave in similar manners. Likewise, we can see similar trends in the purity values (Figure 6.6). If we were presented with a data set from an unknown agent, examining the entropy or purity value and comparing it to known agents might allow us to get a sense for the relative complexity of the unknown agent.

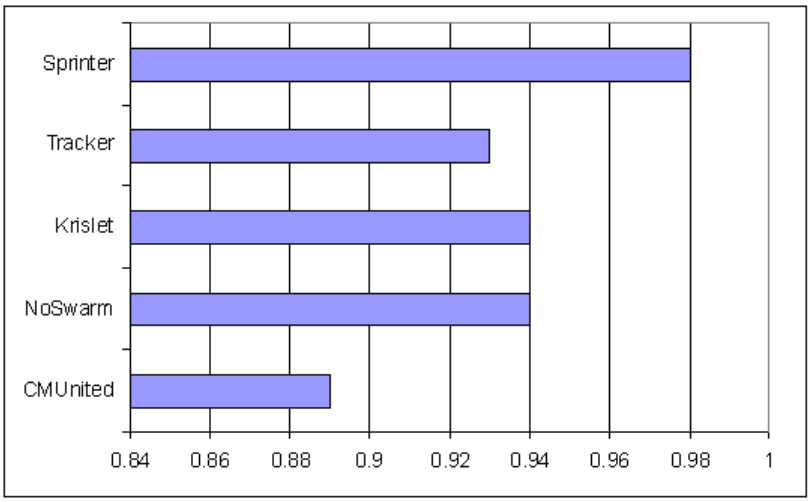


**Figure 6.6:** The purity values for the agents using an initial case base of size 6000.

# Chapter 7

# Prototyping

In case-based reasoning, the current situation is compared to previously encountered situations in the case base in order to find an appropriate solution to the current situation. The ability of a case-based reasoning system to perform well can be related to the number of previously encountered situations that are in the case-base. The more cases in the case base, the more likely we will be able to locate a case that is similar, and has a similar solution, to our current situation. As was discussed in previous chapters, the case base must be searched within a real-time limit so the number of cases contained in the case base are limited.

A way to maximize the information contained in a case-base is to take several cases and replace them with a single case that represents the cases it replaces. This would result in the case base becoming smaller in size, but ideally containing similar information to what it did previously. This process, called prototyping, will allow the information it the case base to remain similar but will reduce the amount of time it takes to search the case base.

### Similar Cases

When collecting training data in an automated and unsupervised fashion, there exists the possibility that some of the collected data items will be quite similar and possibly

identical. In some situations, these similar data items could contain subtle differences that lead to the items belonging to different classes. However, often times these items contain nearly identical information and keeping all of the items is no more beneficial than only keeping one of the items and discarding the rest.

The possibility of having such redundant data is especially important in applications that have real-time constraints and where there is an extra computational cost for each additional data item that is used. If this redundant data provides no noticeable benefit but adds computational cost, the removal of such redundancies could be desirable.

#### Compression

Prototyping a case base can also be used to *compress* the case base. Consider the situation where a case base has been constructed to be searchable withing a real-time limit, $T$. Suppose then that the real-time limit is reduced to $T_{new}$. Using prototyping the case base can be compressed so that it is now searchable within $T_{new}$, but the case base should ideally not lose a significant amount of information.

If prototyping was not used, another method would need to be employed to remove cases from the case base so that it could be searched within the new time limit. The primary advantage of prototyping in this situation is that rather than removing cases it attempts to combine cases, maintaining the information contained in the case base.

## 7.1 Prototyping Methods

In order to decrease the size of the case-base by removing similar cases, there are two major steps that must be performed:

1. Group similar cases together

2. Create a prototypical case from each grouping

The first step, grouping together similar cases, was the focus of Chapter 6. In that chapter the case structure and distance calculation were considered and a variety of clustering algorithms were examined. The clusters of cases produced using these algorithms will be utilized by the second step, creating prototypical cases.

One potentially problematic issue arises when considering clusters that are not *clean*. A *clean* cluster is composed entirely of a single class of case. In an imitative domain, all the cases in a cluster would perform a similar type of action. This situation can be handled in several ways:

- *The non-clean clusters are excluded from the prototyping process.* This method assumes that the non-clean clusters contain important cases. One example could be cases that exist at the boundary between classes so they might be clustered with cases from another class. As such, no prototyping would be performed in order to avoid removing potentially important cases.

- *The non-clean clusters are prototyped..* This method treats the impurities in the clusters as noise or erroneous data. The cases that are part of a different class will be treated as cases from the dominant class when used to create a prototypical case.

This chapter will focus on techniques that can be used to create a prototypical case from a cluster of cases. The following sections will examine several techniques that can be used for prototyping.

### 7.1.1 Using a Case Member

The simplest method for creating a prototypical case from a cluster of cases is to simply use a single case from the cluster, a cluster member, as the prototypical case

and discard the remaining members of the cluster. This method is useful because it does not require creating a new case, but instead it reuses an existing case. By avoiding the creation of a new case the case base is guaranteed to be composed entirely of data that was obtained through direct observation of the agent being imitated.

When using a cluster member as the prototypical case, any of the cases in the cluster could be used. While randomly selecting a case from the cluster is one possible solution, we will examine a more structured approach. Given that the prototypical case should be representative of the entire cluster, the case in the cluster that is *most similar* to the other cases will be used as the prototypical case. For a cluster with $n$ cases in it and containing the cases $\{C_i, \ldots, C_n\}$ , we locate the prototypical case by finding the case that is the minimum distance (where the distance between two cases is $d(C_i, C_j)$) from all other cases in the cluster. If the selected case has a different class than the majority of the cluster, it has its class set to that majority class. The method for selecting the cluster member to use as the prototypical case is shown in Equation 7.1.

$$C_{gen} = \arg \min_{C_i=C_1}^{C_n} \sum_{j=1}^{n} d(C_i, C_j) \tag{7.1}$$

While this technique has the benefit of not having to create new cases, it does have the limitation that significant amounts of information are discarded. Since only the case that becomes the prototypical case is kept, the information contained in the remaining cases is lost when those cases are removed from the case base.

### 7.1.2 Creating an Average Case

The second method of creating a prototypical case from a cluster of cases is to create an "average case". This entails determining an average position that the spatial objects, contained in each case, will be located when examining all cases in the cluster.

Compared to the first method, this method constructs a novel case and does not reuse an existing case. The process of creating an average case when all cases have a fixed number of features is quite simple. The average case would just contain the average value of each feature.

The averaging process becomes more difficult in situations where cases can have a different number of objects in them. For example, consider in the soccer domain when one case contains two teammate objects but another case in the cluster only contained a single teammate object. Performing an "average" then becomes a non-trivial task when the issue of object matching and dealing with extra objects becomes a factor. To deal with these issues we propose Algorithm 8.

**Algorithm 8** Spatial Cluster-Average Prototype

**Inputs:** cluster of cases
**Outputs:** prototypical case

```
SCAP(cluster)
    remove the case, C, that is closest to all other cases
    for(each feature f in C):
       create an empty list and add the f to that list
    end loop
    while(more cases exist in cluster):
       remove the next case in the cluster, N
       for(each feature f in C):
          match f with a feature in N
          add the feature value from N to the list for f
        end loop
    end loop
    create a prototypical case, P, that contains no objects
    for(each feature, f, in C):
      get the list, L, that f belongs to
       compute the average location of all features in L
       add a feature with the average location to P
    end loop
    return P
end
```

This algorithm uses a case in the cluster that is central to the other cases (using Equation 7.1) and performs a pair-wise matching between that case and each of the

other cases in the cluster. The resulting prototype will have the same number of features as the central case and the feature values will be highly dependant on how the other cases matched to the central case. This means changing the case used as the central case can result in different prototype cases being produced. Any case with more objects than the central case will have the remaining objects ignored, while cases with fewer objects then the central case will have no impact on the averaging process for the objects they are missing.

### 7.1.3 Creating a Range Case

Unlike with the previous two approaches, the range case method will create a new case that contains knowledge from all cases in the cluster by creating a range of values that each object may be located in. Ideally, this will minimize the amount of data that is lost in the prototyping process. For example, consider a hypothetical set of cases that are all associated with action $A$. If those cases formed the boundary between cases of action $A$ and cases of action $B$, using the cluster member or cluster average approaches would replace the entire boundary region with a case that contains objects located at specific locations. Instead, the range approach would create a range of possible locations where each object could be located. As long as an object is within the range, it can be thought to be at the same location as the range.

The algorithm for this method is identical to the algorithm for creating an average case except that instead of calculating an average for each object a range is calculated for each object. Any object that is located within the range of values specified by this type of case is said to be an exact match. The main benefit of this method is that it allows for regions where objects of a specific type are likely to occur and does not limit the object to existing at a fixed point. Algorithm 9 details the cluster range method.

The range of values is created by determining the range of locations in each feature

**Algorithm 9** Spatial Cluster-Range Prototype

**Inputs:** cluster of cases
**Outputs:** prototypical case

```
SCRP(cluster)
    remove the case, C, that is closest to all other cases
    for(each feature f in C):
       create an empty list and add the f to that list
    end loop
    while(more cases exist in cluster):
       remove the next case in the cluster, N
       for(each feature f in C):
          match f with a feature in N
          add the feature value from N to the list for f
        end loop
    end loop
    create a prototypical case, P, that contains no objects
    for(each feature, f, in C):
      get the list, L, that f belongs to
       compute the range of locations that exist in L
       add a feature with the range of location to P
    end loop
    return P
end
```

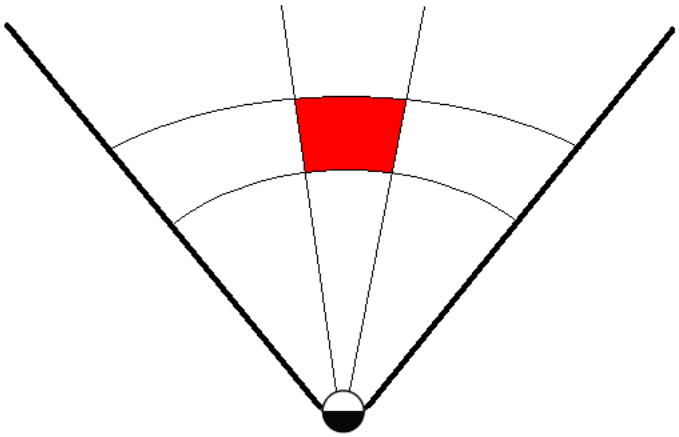

**Figure 7.1:** The range where a particular object can exist.

list, where the location is represented by a distance and direction relative to the agent. The range value contains the minimum and maximum values that the distance and direction values can take. Figure 7.1 shows an agent's field of vision, with the dark region representing the possible range for a particular object.

The primary downside of this approach is the added complexity it adds. Not only will cases be required to support objects at ranges of locations, instead of fixed locations, but the process of object matching also becomes more difficult. For the purpose of object matching, a range of values will be considered to be located at the centre of the range. While this may not be ideal, it allows cases with range values to be compared to cases without range values and does not require more complex object matching process (which would likely require more computational time).

## 7.2 Experimental Results

The experiments in this chapter will examine the various clustering methods we have described and determine the following:

- Do any of the prototyping methods outperform the others?

- What impact do the prototyping methods have on the imitative ability of the agents?

- Should dirty clusters be prototyped or left unchanged?

In the previous chapter, using the distance feature vector approach with k-means clustering was found to be the best clustering method, so the clusters used as input for prototyping algorithms will be generated using that clustering approach. Also, similar to the previous chapter, we will look at reducing a set of *6000*, *12000*, and *18000* cases to the maximum number of cases that can be searched within the real-time constraints. It should be noted that using an initial case base of size *24000* cases is not being examined since, as shown in the previous chapter, a case base that large provided significantly worse results compared to the smaller initial case bases.

### 7.2.1 Results

The results are generated for each team both when dirty clusters are prototyped and when dirty clusters are not prototyped (their cases remain unchanged) and for a variety of initial case base sizes. The results are as follows:

- **Dirty prototyped:**
  - **6000 cases:** Sprinter (Table 7.1), Tracker (Table 7.2), Krislet (Table 7.3), NoSwarm (Table 7.4) and CMUnited (Table 7.5)
  - **12000 cases:** Sprinter (Table 7.6), Tracker (Table 7.7), Krislet (Table 7.8), NoSwarm (Table 7.9) and CMUnited (Table 7.10)
  - **18000 cases:** Sprinter (Table 7.11), Tracker (Table 7.12), Krislet (Table 7.13), NoSwarm (Table 7.14) and CMUnited (Table 7.15)
- **Dirty not prototyped:**

- **6000 cases:** Sprinter (Table 7.16), Tracker (Table 7.17), Krislet (Table 7.18), NoSwarm (Table 7.19) and CMUnited (Table 7.20)
- **12000 cases:** Sprinter (Table 7.21), Tracker (Table 7.22), Krislet (Table 7.23), NoSwarm (Table 7.24) and CMUnited (Table 7.25)
- **18000 cases:** Sprinter (Table 7.26), Tracker (Table 7.27), Krislet (Table 7.28), NoSwarm (Table 7.29) and CMUnited (Table 7.30)

**Table 7.1:** Prototyping - Sprinter - Prototype Dirty Clusters - 6000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 5271 | 0.72(*) +/- 0.007 | — | 0.90 | 0.54 |
| **Cluster Average** | 5271 | 0.74(*) +/- 0.008 | — | 0.92 | 0.56 |
| **Cluster Range** | 5271 | 0.67 +/- 0.008 | — | 0.85 | 0.49 |

**Table 7.2:** Prototyping - Tracker - Prototype Dirty Clusters - 6000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 4409 | 0.78 +/- 0.007 | — | 0.77 | 0.78 |
| **Cluster Average** | 4409 | 0.82(*) +/- 0.009 | — | 0.80 | 0.84 |
| **Cluster Range** | 4409 | 0.75 +/- 0.010 | — | 0.74 | 0.76 |

**Table 7.3:** Prototyping - Krislet - Prototype Dirty Clusters - 6000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 4612 | 0.60 +/- 0.011 | 0.27 | 0.76 | 0.77 |
| **Cluster Average** | 4612 | 0.64(*) +/- 0.008 | 0.32 | 0.76 | 0.84 |
| **Cluster Range** | 4612 | 0.58 +/- 0.006 | 0.23 | 0.72 | 0.79 |

**Table 7.4:** Prototyping - NoSwarm - Prototype Dirty Clusters - 6000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 4727 | 0.65(*) +/- 0.007 | 0.24 | 0.87 | 0.84 |
| **Cluster Average** | 4727 | 0.66(*) +/- 0.009 | 0.27 | 0.85 | 0.86 |
| **Cluster Range** | 4727 | 0.57 +/- 0.009 | 0.19 | 0.82 | 0.70 |

**Table 7.5:** Prototyping - CMUnited - Prototype Dirty Clusters - 6000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 5078 | 0.61 +/- 0.011 | 0.49 | 0.69 | 0.65 |
| **Cluster Average** | 5078 | 0.64(*) +/- 0.008 | 0.52 | 0.77 | 0.63 |
| **Cluster Range** | 5078 | 0.61 +/- 0.010 | 0.44 | 0.76 | 0.63 |

**Table 7.6:** Prototyping - Sprinter - Prototype Dirty Clusters - 12000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 5271 | 0.66 +/- 0.012 | — | 0.88 | 0.44 |
| **Cluster Average** | 5271 | 0.68 +/- 0.010 | — | 0.91 | 0.45 |
| **Cluster Range** | 5271 | 0.64 +/- 0.009 | — | 0.80 | 0.48 |

**Table 7.7:** Prototyping - Tracker - Prototype Dirty Clusters - 12000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 4409 | 0.76 +/- 0.014 | — | 0.75 | 0.77 |
| **Cluster Average** | 4409 | 0.77 +/- 0.011 | — | 0.79 | 0.75 |
| **Cluster Range** | 4409 | 0.73 +/- 0.009 | — | 0.71 | 0.75 |

**Table 7.8:** Prototyping - Krislet - Prototype Dirty Clusters - 12000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 4612 | 0.58 +/- 0.009 | 0.24 | 0.75 | 0.75 |
| **Cluster Average** | 4612 | 0.60 +/- 0.013 | 0.25 | 0.75 | 0.80 |
| **Cluster Range** | 4612 | 0.55 +/- 0.012 | 0.14 | 0.75 | 0.76 |

**Table 7.9:** Prototyping - NoSwarm - Prototype Dirty Clusters - 12000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 4727 | 0.62 +/- 0.009 | 0.20 | 0.84 | 0.82 |
| **Cluster Average** | 4727 | 0.60 +/- 0.010 | 0.22 | 0.73 | 0.85 |
| **Cluster Range** | 4727 | 0.55 +/- 0.008 | 0.16 | 0.78 | 0.71 |

**Table 7.10:** Prototyping - CMUnited - Prototype Dirty Clusters - 12000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 5078 | 0.58 +/- 0.009 | 0.50 | 0.61 | 0.63 |
| **Cluster Average** | 5078 | 0.61 +/- 0.011 | 0.51 | 0.70 | 0.62 |
| **Cluster Range** | 5078 | 0.56 +/- 0.008 | 0.40 | 0.68 | 0.60 |

**Table 7.11:** Prototyping - Sprinter - Prototype Dirty Clusters - 18000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 5271 | 0.60 +/- 0.013 | — | 0.84 | 0.36 |
| **Cluster Average** | 5271 | 0.63 +/- 0.010 | — | 0.86 | 0.40 |
| **Cluster Range** | 5271 | 0.59 +/- 0.009 | — | 0.77 | 0.41 |

**Table 7.12:** Prototyping - Tracker - Prototype Dirty Clusters - 18000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 4409 | 0.72 +/- 0.008 | — | 0.71 | 0.73 |
| **Cluster Average** | 4409 | 0.71 +/- 0.011 | — | 0.72 | 0.70 |
| **Cluster Range** | 4409 | 0.68 +/- 0.010 | — | 0.64 | 0.72 |

**Table 7.13:** Prototyping - Krislet - Prototype Dirty Clusters - 18000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 4612 | 0.55 +/- 0.013 | 0.20 | 0.72 | 0.73 |
| **Cluster Average** | 4612 | 0.57 +/- 0.010 | 0.23 | 0.71 | 0.77 |
| **Cluster Range** | 4612 | 0.52 +/- 0.012 | 0.12 | 0.68 | 0.76 |

**Table 7.14:** Prototyping - NoSwarm - Prototype Dirty Clusters - 18000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 4727 | 0.60 +/- 0.009 | 0.18 | 0.82 | 0.80 |
| **Cluster Average** | 4727 | 0.59 +/- 0.011 | 0.21 | 0.77 | 0.79 |
| **Cluster Range** | 4727 | 0.53 +/- 0.010 | 0.14 | 0.78 | 0.67 |

**Table 7.15:** Prototyping - CMUnited - Prototype Dirty Clusters - 18000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 5078 | 0.55 +/- 0.013 | 0.42 | 0.60 | 0.63 |
| **Cluster Average** | 5078 | 0.58 +/- 0.011 | 0.48 | 0.67 | 0.59 |
| **Cluster Range** | 5078 | 0.54 +/- 0.009 | 0.37 | 0.65 | 0.60 |

**Table 7.16:** Prototyping - Sprinter - Do Not Prototype Dirty Clusters - 6000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 5280 | 0.72(*) +/- 0.004 | — | 0.90 | 0.54 |
| **Cluster Average** | 5280 | 0.74(*) +/- 0.006 | — | 0.92 | 0.56 |
| **Cluster Range** | 5280 | 0.67 +/- 0.006 | — | 0.85 | 0.49 |

**Table 7.17:** Prototyping - Tracker - Do Not Prototype Dirty Clusters - 6000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 4493 | 0.78 +/- 0.008 | — | 0.78 | 0.78 |
| **Cluster Average** | 4493 | 0.82(*) +/- 0.006 | — | 0.80 | 0.84 |
| **Cluster Range** | 4493 | 0.76 +/- 0.006 | — | 0.71 | 0.76 |

**Table 7.18:** Prototyping - Krislet - Do Not Prototype Dirty Clusters - 6000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 4674 | 0.60 +/- 0.010 | 0.27 | 0.77 | 0.77 |
| **Cluster Average** | 4674 | 0.64(*) +/- 0.008 | 0.32 | 0.76 | 0.84 |
| **Cluster Range** | 4674 | 0.58 +/- 0.007 | 0.23 | 0.72 | 0.80 |

**Table 7.19:** Prototyping - NoSwarm - Do Not Prototype Dirty Clusters - 6000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 4786 | 0.66(*) +/- 0.007 | 0.24 | 0.88 | 0.86 |
| **Cluster Average** | 4786 | 0.66(*) +/- 0.005 | 0.27 | 0.85 | 0.86 |
| **Cluster Range** | 4786 | 0.57 +/- 0.007 | 0.19 | 0.83 | 0.70 |

**Table 7.20:** Prototyping - CMUnited - Do Not Prototype Dirty Clusters - 6000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 5165 | 0.62 +/- 0.013 | 0.50 | 0.69 | 0.67 |
| **Cluster Average** | 5165 | 0.64(*) +/- 0.008 | 0.52 | 0.77 | 0.64 |
| **Cluster Range** | 5165 | 0.61 +/- 0.009 | 0.44 | 0.76 | 0.63 |

**Table 7.21:** Prototyping - Sprinter - Do Not Prototype Dirty Clusters - 12000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 6195 | 0.73(*) +/- 0.009 | — | 0.88 | 0.58 |
| **Cluster Average** | 6195 | 0.74(*) +/- 0.010 | — | 0.91 | 0.57 |
| **Cluster Range** | 6195 | 0.69 +/- 0.006 | — | 0.84 | 0.54 |

**Table 7.22:** Prototyping - Tracker - Do Not Prototype Dirty Clusters - 12000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 6012 | 0.79(*) +/- 0.008 | — | 0.79 | 0.79 |
| **Cluster Average** | 6012 | 0.83(*) +/- 0.009 | — | 0.82 | 0.84 |
| **Cluster Range** | 6012 | 0.78 +/- 0.009 | — | 0.75 | 0.81 |

**Table 7.23:** Prototyping - Krislet - Do Not Prototype Dirty Clusters - 12000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 5587 | 0.61(*) +/- 0.007 | 0.29 | 0.78 | 0.76 |
| **Cluster Average** | 5587 | 0.64(*) +/- 0.007 | 0.33 | 0.76 | 0.83 |
| **Cluster Range** | 5587 | 0.59 +/- 0.008 | 0.27 | 0.73 | 0.77 |

**Table 7.24:** Prototyping - NoSwarm - Do Not Prototype Dirty Clusters - 12000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 5698 | 0.66(*) +/- 0.006 | 0.25 | 0.87 | 0.86 |
| **Cluster Average** | 5698 | 0.68(*) +/- 0.007 | 0.29 | 0.91 | 0.84 |
| **Cluster Range** | 5698 | 0.59 +/- 0.007 | 0.24 | 0.82 | 0.71 |

**Table 7.25:** Prototyping - CMUnited - Do Not Prototype Dirty Clusters - 12000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 6406 | 0.63(*) +/- 0.009 | 0.52 | 0.70 | 0.67 |
| **Cluster Average** | 6406 | 0.65(*) +/- 0.010 | 0.53 | 0.76 | 0.66 |
| **Cluster Range** | 6406 | 0.61 +/- 0.014 | 0.42 | 0.78 | 0.63 |

**Table 7.26:** Prototyping - Sprinter - Do Not Prototype Dirty Clusters - 18000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 8247 | 0.75(*) +/- 0.007 | — | 0.90 | 0.60 |
| **Cluster Average** | 8247 | 0.76(*) +/- 0.006 | — | 0.91 | 0.61 |
| **Cluster Range** | 8247 | 0.70 +/- 0.009 | — | 0.86 | 0.54 |

**Table 7.27:** Prototyping - Tracker - Do Not Prototype Dirty Clusters - 18000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 8477 | 0.81(*) +/- 0.007 | — | 0.83 | 0.79 |
| **Cluster Average** | 8477 | 0.85(*) +/- 0.009 | — | 0.84 | 0.86 |
| **Cluster Range** | 8477 | 0.78 +/- 0.005 | — | 0.74 | 0.82 |

Values of $f1_{global}$ marked with a (*) denoted values that are statistically significant improvements (p=0.01) over the values found in Table 5.2.

### 7.2.2 Analysis and Conclusions

The results using prototyping, which attempt to reduce case bases of size *6000*, *12000* and *18000* to the real-time maximum size, were compared to the results found in Chapter 5 when using a case base of the real-time maximum size without any pre-processing (Table 5.2).

Using an initial case base of size *6000*, we can see that using a case base prototyped with the cluster average method performs statistically significantly better

**Table 7.28:** Prototyping - Krislet - Do Not Prototype Dirty Clusters - 18000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 7691 | 0.63(*) +/- 0.008 | 0.33 | 0.80 | 0.76 |
| **Cluster Average** | 7691 | 0.65(*) +/- 0.010 | 0.34 | 0.78 | 0.83 |
| **Cluster Range** | 7691 | 0.60 +/- 0.012 | 0.27 | 0.72 | 0.81 |

**Table 7.29:** Prototyping - NoSwarm - Do Not Prototype Dirty Clusters - 18000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 7723 | 0.68(*) +/- 0.009 | 0.29 | 0.90 | 0.85 |
| **Cluster Average** | 7723 | 0.70(*) +/- 0.007 | 0.33 | 0.89 | 0.88 |
| **Cluster Range** | 7723 | 0.61 +/- 0.011 | 0.26 | 0.85 | 0.72 |

than using no prototyping. While the cluster average method was found to be an improvement for all data sets, the cluster member approach was only found to be a statistical improvement for the Sprinter and NoSwarm teams. The cluster range method was never found to be an improvement, and in some situations even resulted in a decrease in performance (with Sprinter, Tracker and NoSwarm). We can see from these results that compressing a case base of size 6000 to a smaller size resulted in improved performance compared to simply using a smaller sized case base to begin with. While the compression process decreased performance in some cases, most notably the cluster range method, overall the ability to compress a larger case base to a smaller size proved beneficial. The information contained in *6000* cases was transfered to the smaller size, in general, without a significant loss.

**Table 7.30:** Prototyping - CMUnited - Do Not Prototype Dirty Clusters - 18000 Cases

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **Cluster Member** | 8741 | 0.64(*) +/- 0.011 | 0.54 | 0.73 | 0.65 |
| **Cluster Average** | 8741 | 0.66(*) +/- 0.010 | 0.54 | 0.78 | 0.66 |
| **Cluster Range** | 8741 | 0.62(*) +/- 0.009 | 0.50 | 0.75 | 0.61 |

One unexpected result was the overall poor performance of the cluster range approach. It was thought that the range of possible positions for objects would help to model regions where objects were likely to exist. One explanation for this poor performance is that there is a relation between the position of objects in a case. For example, several cases may contain a ball and goal object. In all of the cases, the ball and goal may be in different positions leading to a fairly large range of values for their locations. However, there may be some unknown constraint expressed in the cases, such as the ball is always a minimum distance from the goal. Creating ranges of values in a prototypical case may result in cases that *break* this unknown constraint being found similar to the prototypical case. This leads us to conclude that representing objects as being at specific locations in preferable to ranges of possible locations.

Both cluster member and cluster average approaches were found to be beneficial, however since the cluster average method was found to show improved results for *all* agent types it will be selected as the prototyping method to use in subsequent chapters. Ideally, the algorithm should work for a variety of data sets so that a single algorithm can be employed without knowledge of the domain or requiring comparison between potential algorithms.

Comparing the two approaches for handling dirty clusters, either prototyping them or leaving them unchanged, we see almost no difference in the results when compressing *6000* cases but differences are noticeable for larger initial case bases. The primary difference is in the size of the prototyped case base. While the approach that prototypes dirty clusters always has the desired case base size, the other approach leads to case bases that are larger. This is because the cases in a dirty cluster remain unchanged, so instead of providing a single prototype per cluster all members of the cluster are added to the case base. This becomes more noticeable as the size of the initial case base is increased. This makes it more difficult to specify an exact size for the resulting case base. For example, when an initial case base of size *18000* is used the resulting prototyped case bases are nearly double their desired value. This would require further preprocessing to reduce these case bases to sizes that can be used within a real-time limit. While the case bases that do not prototype dirty clusters perform better than the case bases that do prototype dirty clusters this is an unfair comparison. The case bases that do not prototype dirty cluster have more cases in them, which provides them with more data to use but also causes them to miss their real-time deadlines. Given that prototyping dirty clusters provides more control over the final case base size, this approach will be used for subsequent experiments.

# Chapter 8

# Combined Techniques

This chapter does not attempt to examine any new preprocessing techniques but instead looks at how feature reduction and prototyping can be used together. At first, this may seem like a trivial point and a solution would be to simply perform both methods. Both method will be applied, but this chapter will examine the impact of the order of preprocessing.

Each of the preprocessing methods, feature reduction and prototyping, will modify the case base in some way such that the case base will be different after the preprocessing has occurred. The change to the case base may have an impact on how the next preprocessing method performs. We will examine the following two preprocessing orders:

- **Feature removal before prototyping:** Feature removal will decrease the number of features the clustering algorithms use to cluster the cases and may combine some clusters together. This would lead to the removal of more cases during the prototyping process if the clusters were larger.

- **Feature removal after prototyping:** Since the prototyping process will even prototype cluster that are not clean, meaning they contain cases related to more than one action type, some erroneous cases (such as mislabeled cases) may be

removed from the case base. This might be beneficial for the feature removal algorithms since improperly labeled training or testing cases could result in misleading performance metrics. For example, the *f1* value might seem lower if a retrieved case appeared to be of the wrong class but was actually just mislabeled.

The remainder of this chapter will experimentally examine these two orderings and determine if the order of preprocessing matters and if a specific ordering is particularly beneficial.

## 8.1 Experimental Results

In the previous chapters, we have determined the best methods for feature reduction and prototyping a case base. In summary, the following algorithms were found to perform best and will be used during this set of experiments:

- **Feature Removal:** Binary feature weighting with a dynamic training set
- **Clustering:** Distance feature vector method with k-means clustering
- **Prototyping:** Cluster average prototyping

The goal of these experiments is to perform both feature reduction and prototyping to a case base and to examine if the order in which they are performed makes a noticeable difference. The following two situations will be tested:

- **Feature removal followed by prototyping (FR-P):** Feature removal will be performed initially, removing the features that were not found to be beneficial. Next, the entire case base will have 10% of the cases removed using prototyping. In this situation, the prototyping will work on data with less features.

- **Prototyping followed by feature removal (P-FR):** The entire case base will be reduced by 10% using prototyping. Feature removal will then be performed. Instead of performing feature removal on the original case base, feature removal will be performed on a prototyped case base which could change the features that are selected to be removed.

Following this preprocessing phase a case base of the maximum allowable size, based on the agent and the number of features remaining, will be randomly selected from the entire preprocessed case base. Therefore, assuming both methods find similar feature weights they will produce training sets of the same size.

### 8.1.1 Results

The results for Sprinter (Table 8.1), Tracker (Table 8.2), Krislet (Table 8.3), NoSwarm (Table 8.4) and CMUnited (Table 8.5) are found in the following tables.

**Table 8.1:** Combined Preprocessing - Sprinter

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **FR-P** | 36897 | 0.94(*) +/- 0.003 | — | 0.95 | 0.93 |
| **P-FR** | 36897 | 0.92 +/- 0.005 | — | 0.95 | 0.90 |

**Table 8.2:** Combined Preprocessing - Tracker

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **FR-P** | 30863 | 0.99 +/- 0.001 | — | 0.99 | 0.99 |
| **P-FR** | 30863 | 0.98 +/- 0.001 | — | 0.98 | 0.98 |

Values of $f1_{global}$ marked with (*) denote a statistically significant (p=0.01) increase over using a dynamic training set with a binary feature selection algorithm.

**Table 8.3:** Combined Preprocessing - Krislet

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **FR-P** | 32284 | 0.92(*) +/- 0.007 | 0.84 | 0.96 | 0.96 |
| **P-FR** | 32284 | 0.87 +/- 0.005 | 0.72 | 0.95 | 0.94 |

**Table 8.4:** Combined Preprocessing - NoSwarm

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **FR-P** | 33089 | 0.89(*) +/- 0.004 | 0.80 | 0.94 | 0.93 |
| **P-FR** | 33089 | 0.85 +/- 0.007 | 0.71 | 0.93 | 0.89 |

### 8.1.2 Analysis and Conclusions

In these results, we see that performing feature selection before prototyping leads to a statistically significant increase over the previous best results that we have seen thus far (using a dynamic training set and binary feature selection). For every type of agent, except Tracker, the $f1_{global}$ metric showed a significant increase. However, when prototyping was performed first there was only a statistically significant increase for the CMUnited data. For the other data sets there was a slight improvement, but not a statistically significant one. It should be noted that there was no *decrease* when using prototyping first, just no increase.

**Table 8.5:** Combined Preprocessing - CMUnited

| | **Cases** | $f1_{global}$ | $f1_{kick}$ | $f1_{dash}$ | $f1_{turn}$ |
|---|---|---|---|---|---|
| **FR-P** | 17773 | 0.75(*) +/- 0.011 | 0.73 | 0.80 | 0.72 |
| **P-FR** | 17773 | 0.73(*) +/- 0.008 | 0.70 | 0.76 | 0.73 |

A likely reason for these results is that the feature selection greatly reduces the number of features for the prototyping algorithm to work with. To begin with, the clustering algorithm only has to cluster using features that were found to be important. The extra features were removed, so the clustering algorithm is more accurately able to calculate the distance between cases and group similar cases together. In turn, the clusters provided to the prototyping algorithms are likely better so the prototyping algorithm is able to produce better prototypical cases.

On the other hand, when prototyping occurs first it is still working will all of the features and does not get the benefits mentioned above. Thus, prototyping is not able to make as significant an impact on the case base. The feature removal algorithm will then use a case base that is very similar to the original case base, so it is intuitive that we should not expect a significant increase in results.

We can then conclude that, in general, it is desirable to perform feature selection before prototyping. While performing the preprocessing in the reverse order, with prototyping occurring first, will not be a detriment to the case base, the improvement will be less then if prototyping is performed last.

# Chapter 9

# Agent Imitation Framework

One of the primary goals of our case-based imitation work is to develop general purpose techniques for imitating the behaviour of software agents. Currently, as described in this thesis, we have only applied case-based imitation to simulated RoboCup soccer but our aim is to explore a variety of domains, both simulated and physical. Ideally, these techniques should be easily transferable to new domains without making changes to the underlying software structure. The software used in previous case-based imitation work [21,31,32] had several key limitations that prevented it from easily being transferred to different domains.

- **Tightly coupled to RoboCup**: The algorithms used for case-based reasoning and imitation were often tightly coupled with the software used to connect to the soccer server as a RoboCup agent. Changing domains would require modifying source code in numerous areas of the software.

- **Fixed set of sensory objects**: The number and types of objects that the software could handle were hard coded to a fixed set of RoboCup soccer specific objects. Even in the RoboCup domain, adding or removing an object type would require numerous changes to the majority of the algorithms used by the software.

- **Only spatial feature types are supported**: No method was in place for introducing non-spatial features since only a single distance calculation method was available. For example, it would not be possible to add a non-spatial feature, such as the game score or time, since a spatial distance calculation would not be applicable to those feature types.

To overcome these limitations we present the Java Imitation Framework for Spatially-Aware Agents (JIFSA) [19]. This framework is domain independent and will allow us to examine agent imitation in a variety of environments, both simulated and physical. The remained of this chapter will detail how JIFSA overcomes the limitations of existing case-based imitation software.

## 9.1 The Agent

At the heart of the JIFSA imitation framework is the Agent class. This class is used to perform the imitation process and operates in the same environment as the software agent it is imitating. Given that the Agent class should ideally be interchangeable with the original software agent, it accepts similar inputs and produces similar outputs (Figure 9.1). The remainder of this section will describe the internal structure of the Agent class, with Section 9.2 detailing how the inputs and outputs are processed by the system.

### 9.1.1 Internals of the Agent

The Agent class, with its class diagram shown in Figure 9.2, contains the case-based reasoning and imitation algorithms used by the system. The primary method of this class, used to produce output given inputs, is the *senseEnvironment* method. Whenever a new set of inputs, in the form of an *AgentInputs* object, is received from

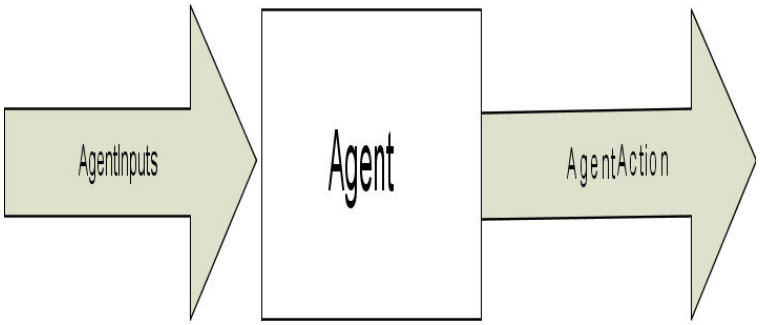


**Figure 9.1:** The imitative agent receiving inputs and producing outputs.

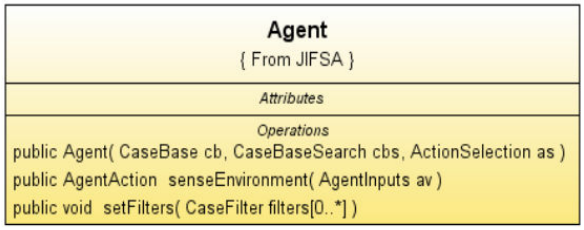


**Figure 9.2:** Class diagram for the Agent class.

the environment the *senseEnvironment* method is used to produce an output action, in the form of an *AgentAction* object. When deployed in its environment, the Agent will continuously have the *senseEnvironment* method invoked, allowing it to interact with its environment.

As shown in the class diagram, the constructor method for an Agent requires three parameters: a *CaseBase*, *CaseBaseSearch* and *ActionSelection*. These classes are modelled after the central case-based imitation algorithms and data structures that were discussed earlier in Chapter 4. However, more focus will be placed on the range of algorithms and data that can be used instead of the specific ones that were used during experimentation.

### Case Base

The central data structure used by the case-based imitation agent, like any case-based reasoning application, is the *case base*. Essentially, the case base is simply a collection

of *cases* as can be seen in Figure 9.3.

The cases in a case base represent the knowledge of the imitation agent, and as such the ability of the agent to imitate is directly tied to the quality of cases in the case base. As we have discussed earlier, in case-based reasoning cases represent previously encountered problems and the solutions to those problems. In case-based imitation, the problem is represented by the sensory inputs received by the agent. In response to these inputs, the agent then performs the solution: a number of actions. Therefore, each case is composed of sensory inputs and actions as shown in Figure 9.4.

As was stated at the start of this chapter, a major motivation for this imitation framework was to allow new sensory inputs to easily be added to a case. The software previously used for case-based imitation work was tightly coupled with RoboCup and cases could only contain the objects visible to a RoboCup agent. Therefore, it was not possible for a case to contain any non-visual inputs like sounds, time, or game score. Although the current imitation work, as described in this thesis, still only makes use of visual inputs, it is now possible to include other types of inputs. This property of the framework will make it possible to attempt case-based imitation in domains outside of RoboCup.

The sensory inputs, in a case, are a collection of features as shown in Figure 9.5. Thus, a case can be composed of all of the input features that an agent can sense in its environment. For example, in the RoboCup domain features could contain information about objects and their distance and direction relative to the agent. Likewise, another feature could contain information about the current *time*. These various features would be *subclasses* of the *Feature* class and would extend that class to contain information about the specific type of feature. Going back to the previous example, a visible object would extend *Feature* to contain positional information.

One reason why, in the previous imitation software, it was difficult to incorporate

new feature types is because there was only one method for *comparing* features. Since all features represented objects, with locations relative to the agent, the comparison simply calculated the *distance* between the objects. Thus, only features that could be represented by spatial positions could be compared. In the current framework, instead of a single method of feature comparison each type of feature can have a separate method of comparison. For each subclass of *Feature*, the comparison method can be set statically for the entire class. If a specific subclass does not have a comparison method explicitly set it instead uses the comparison method of its superclass. This allows each type of feature to have a comparison method that is specifically tailored to itself.

### Case Base Search

The imitation agent has its knowledge stored in the form of a case base, but during run-time it must use that knowledge to determine what actions to perform. Given that the case base contains past problem-solution pairs, the agent must search the case base to find solutions to novel problems. The current problem faced by the agent, its current sensory inputs, is used as a query to search the case base. The cases retrieved when searching the case base can then have their solutions (agent actions) reused to solve the current problem.

The CaseBaseSearch interface, as shown in Figure 9.6, defines a method, *findClosest*, that searches a case base, *possibleCases*, for cases that are close to an input case, *currentCase*. While any search method can be used, the search method used in this thesis is k-nearest neighbour search. For each input case, the distance between the input case and all cases in the case base is calculated. The $k$ cases that are closest to the input case are then returned. This process is shown in Figure 9.7.

One area which has yet to be discussed is how the k-nearest neighbour search calculates the distance between two cases, using the *pairwiseDistance* function in

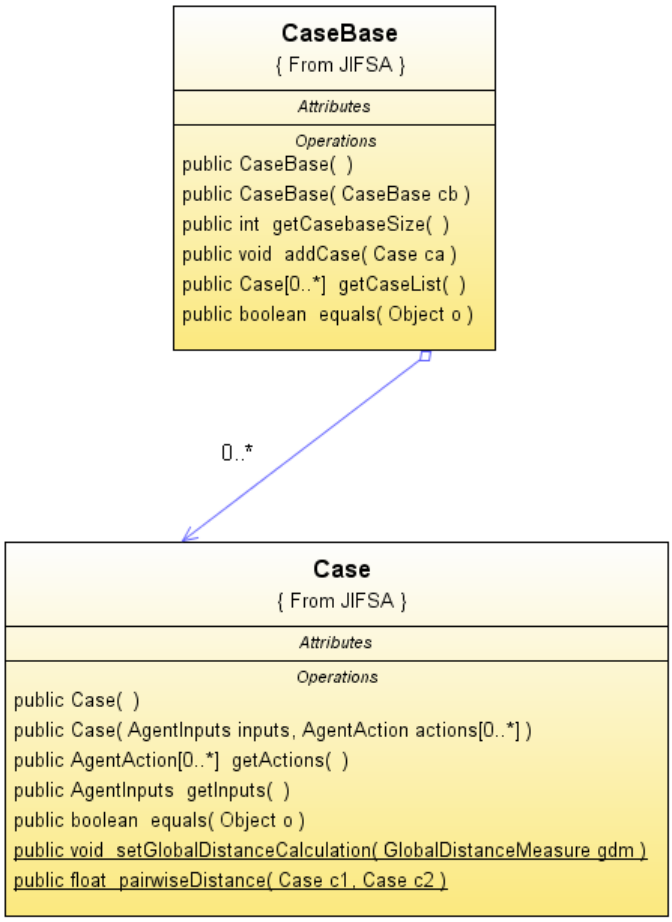


**Figure 9.3:** Class diagram for the CaseBase class.

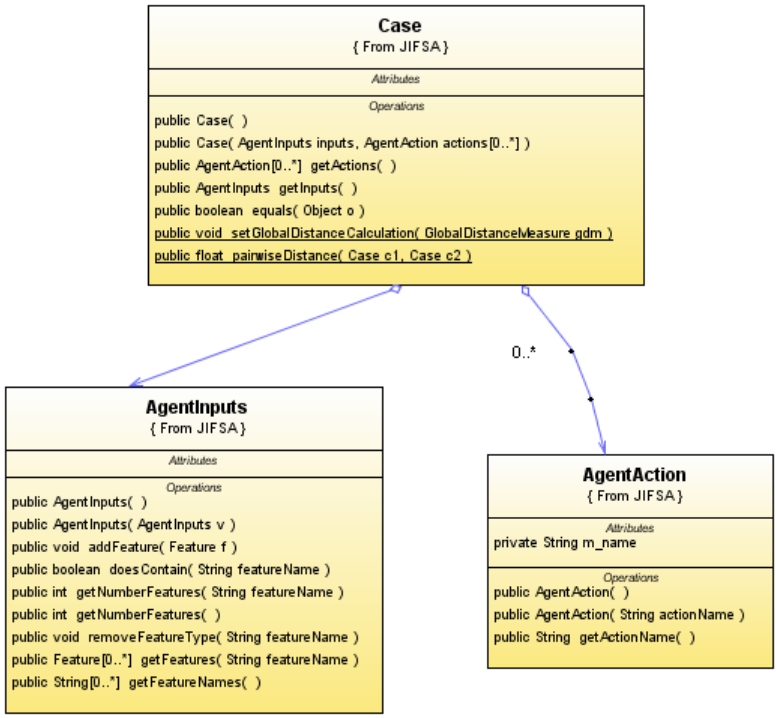


**Figure 9.4:** Class diagram for the Case class.

Figure 9.7. As we described in more detail in Chapter 4, the distance between cases is essentially a function of the distances between individual features. Features from one case are matched with similar features from the other case, and then the distance between those features can be calculated using the comparison method for that type of feature (as described previously). All of the feature distances can then be combined to determine the distance between the cases.

### Action Selection

When searching the case base, one or more potential cases can be retrieved. In Chapter 4, the issue of action selection was described, although throughout this thesis no action selection was needed since only a single case was retrieved from the case base search (a 1-nearest neighbour search). If any case base search is used that can return multiple cases from the case base (multiple candidate solutions) then some method of

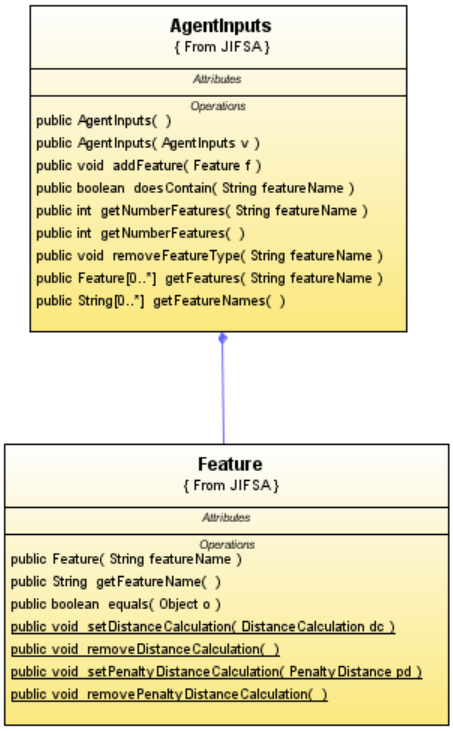


**Figure 9.5:** Class diagram for the AgentInputs class.

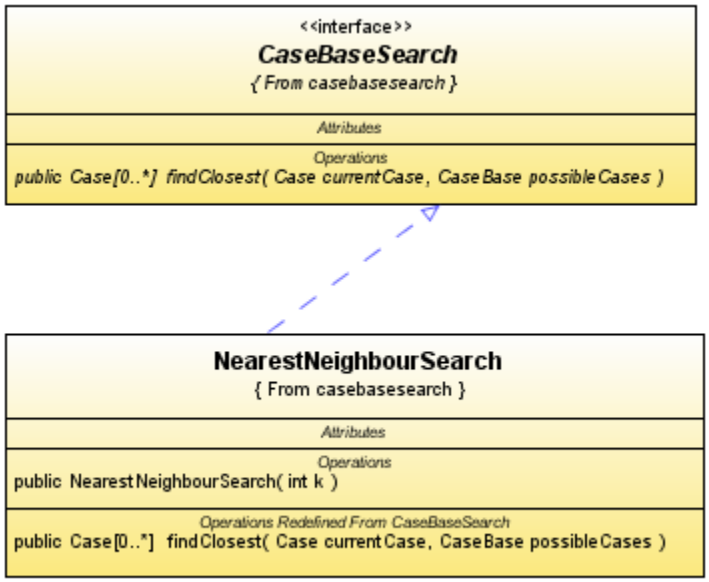


**Figure 9.6:** Class diagram for the CaseBaseSearch interface.

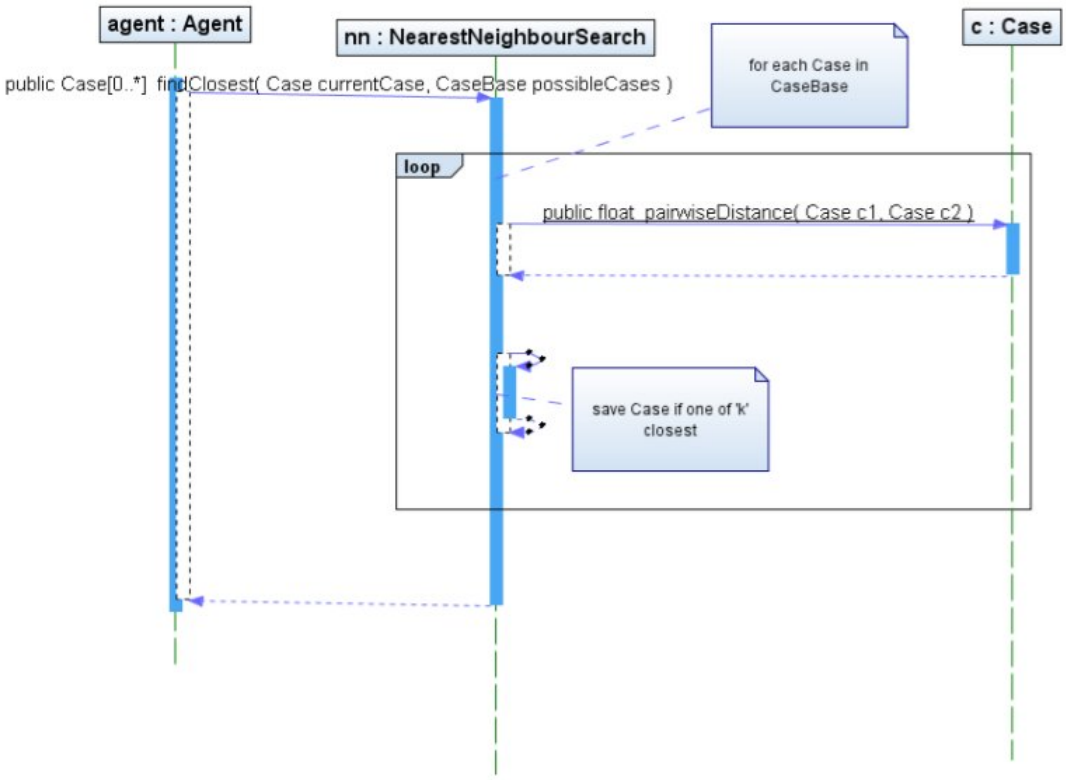


**Figure 9.7:** Sequence diagram for searching the case base using k-nearest neighbour search.

selecting which action to perform is required. Some potential action selection methods are:

- Using the most common action from the returned cases
- Randomly selecting an action from the actions contained in the returned cases
- Using the action from the most similar returned case
- A combination or variation of one or more actions from the returned cases

## 9.2 Inputs and Outputs

So far, we have described the case-based imitation agent and how it uses sensory inputs to determine which actions to perform. This agent, using the JIFSA framework, is designed in such a way that it has no direct knowledge of how to interact with its environment. Although the case base will contain domain specific information, the agent itself will not have any domain information. For example, while a RoboCup imitator would have a case base containing RoboCup specific object types and actions, the agent would not be aware it was involved in a RoboCup soccer game.

Thus, in order to deploy an imitative agent in an environment two things are needed: a *sensory system* and *action execution module*. We previously said, in Chapter 5, what processes must be completed within one RoboCup cycle (shown again in Figure 9.8). Notice how the sensory system corresponds to the *Identify Visible Objects* block and action execution module corresponds to the *Perform Action* block. The sensory system is used to convert sensory data into objects that can be processed by the imitative agent. For example, in simulated RoboCup soccer this involves parsing messages from the server and creating *Feature* objects based on those messages. Likewise, in a physical robot this might involve converting raw sensory data into *Feature*

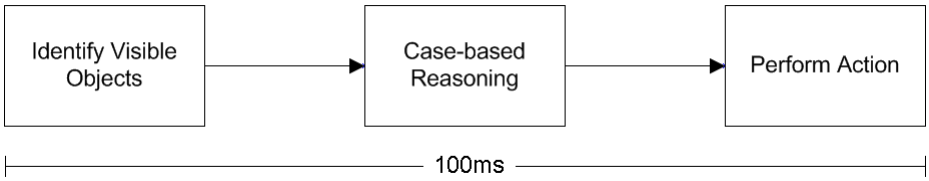


**Figure 9.8:** The activities the RoboCup agent must perform in one 100ms cycle.

objects. These *Feature* objects are then used to create an *AgentInputs* object that can be given as input to the Agent, as shown in Figure 9.9, using the *senseEnvironment* method.

The Agent can then filter the *AgentInputs* object in order to remove unwanted features or otherwise transform the input, search the case base for cases similar to the *AgentInputs* object and return an *AgentAction* to perform (possibly using an action selection algorithm if multiple potential cases are found). It then becomes necessary for the action execution module to actually perform the *AgentAction*. In simulated RoboCup soccer, this involves creating the appropriate message to send to the server. In a physical robot it could involve sending control signals to the various components of the robot.

As we can see, the sensory system and action execution module require direct interaction with physical environment. Therefore, these two areas are domain specific and are unique for each domain the case-based imitator is deployed in. This allows a clear decoupling between the domain independent code contained in JIFSA (Agent, CaseBase, etc.) and any domain specific code (sensory system, action execution module, code to create cases by observing an agent, etc.).

## 9.3 RoboCup Simulation Example

Thus far we have described the JIFSA framework but have have not gone into detail about how it can be deployed in a specific domain. This section will detail how

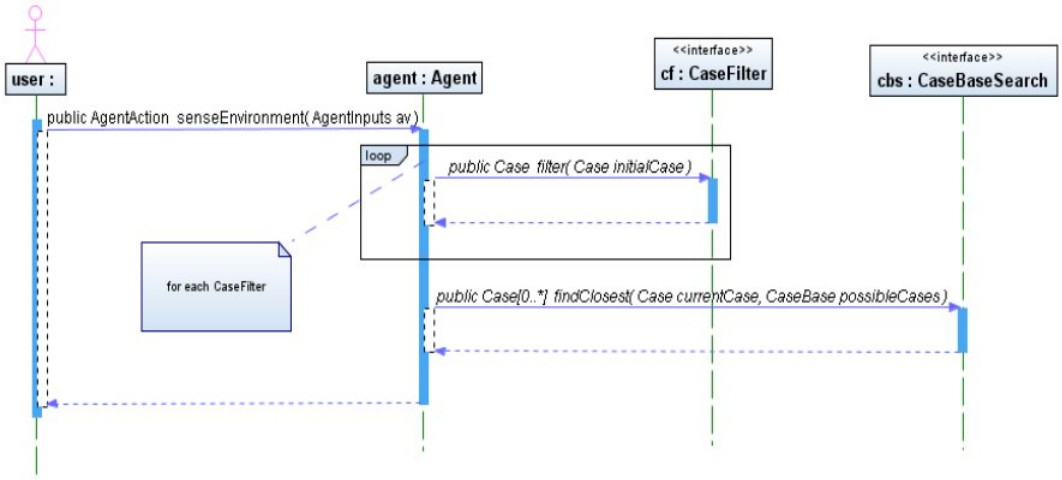


**Figure 9.9:** Sequence diagram for the Agent.

JIFSA was used in the simulated RoboCup domain and should provide insight into how JIFSA can be used in other domains.

In order to use JIFSA in RoboCup, in a software package called RCSImitate (RoboCup Soccer Imitate), the following steps were needed:

- **Sensory Input Definition:** This stage involved defining which types of sensory inputs were detectable in the domain and modelling those sensory inputs as *Feature* objects. In RoboCup, this involved determining which objects the soccer server informs a player about as well as what properties each type of object has. In simulated domains this information is usually presented in a manual or API, whereas in a robotic system it would be dependant of the complexity of the computer vision system used. A sample RoboCup object is shown in Figure 9.10.

- **Action Definition:** Just as the sensory items must be defined, so too must the actions that the agent can perform. In simulated environments, like RoboCup, actions are also often defined in a manual or API whereas in a robotic system

the actions depend on the physical abilities of the robot. A sample RoboCup action is shown in Figure 9.11.

- **Agent Observation and Case Creation:** In order to build the case base, a method must be in place to monitor the behaviour of an agent in response to its environment. In RoboCup, this is done using a proxy utility that is placed between the soccer agent and the server and logs all communication between them [37] (as described in Chapter 4). The communications logs are then parsed and create cases composed of the sensory input and action objects that were previously defined.

- **Sensory System:** This system, derived from code from the Krislet agent, involves communicating with the RoboCup soccer server and parsing messages received from the server. These parsed messages are converted to *AgentInputs* objects and then passed along to the *Agent*.

- **Action Execution Module:** After the Agent has been given an *AgentInputs* from the sensory system it will produce an *AgentAction*. This module, which is also derived from Krislet code, converts the *AgentAction* object into a properly formated message and sends that message back to the soccer server.

As we can see, deploying JIFSA in a novel domain requires an initial definition of sensory inputs and actions followed by several systems for agent observation and environment interaction. These systems all either make use of the core JIFSA *Agent* or extend the data structure used to form a case. Thus, we feel JIFSA can be applied in a variety of domains with a minimal amount of expert knowledge.

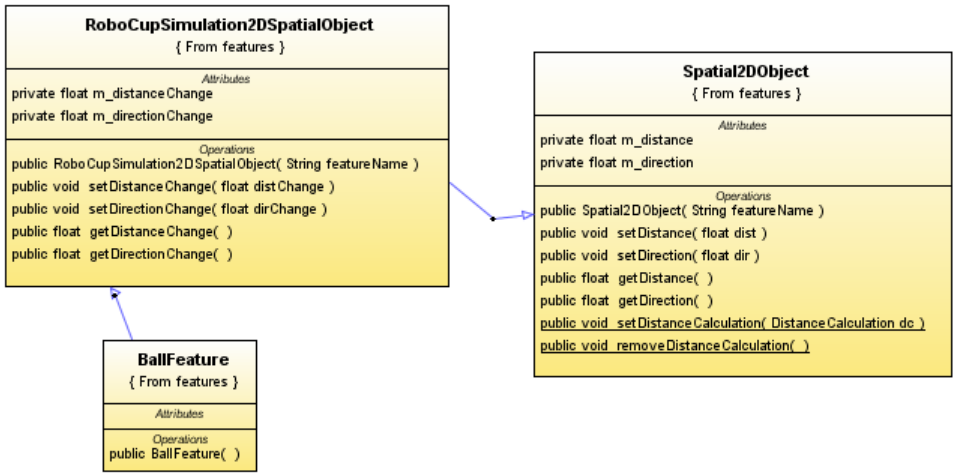


**Figure 9.10:** Class diagram for the Ball feature in RoboCup.

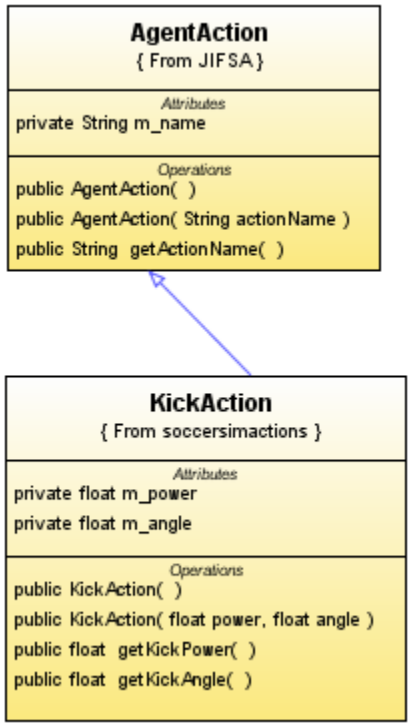


**Figure 9.11:** Class diagram for the Kick action in RoboCup.

## 9.4 Preprocessing the Case Base

Preprocessing plays an important role in the transition of raw cases obtained through observation to cases usable by an imitative agent. While preprocessing cases is not a necessity, the experiments in this thesis have shown preprocessing can help to increase the imitative ability of an agent imitating a RoboCup soccer player. Initially cases are generated through direct observation. A series of one or more preprocessing steps can then be applied, as shown in Figure 9.12, to produce a final case base that is used by the imitative agent.

In Figure 9.12, we see that several different preprocessing steps, $P_1$ to $P_n$, are applied to the the case base. Each of these preprocessing steps produces a new case base that is then used as input to the next preprocessing step. After all $n$ preprocessing steps a new case base is produced that is the result of all $n$ preprocessing steps. A user could potentially create a preprocessing workflow such as this that could automatically apply a series of preprocessing steps on any newly acquired case bases. In fact, the same preprocessing step could be used several times. For example, the data could be prototyped, then features removed and finally prototyped again.

In this thesis, we have examined two methods of preprocessing: feature removal and prototyping. Recall the following:

- **Feature Removal:** a feature selection algorithm is used to select relevant features and then any unused features are removed from thte cases. The result of this preprocessing step will have the same number of cases as the input case base, but each case may be composed of fewer features.

- **Prototyping:** the case base is first clustered using a clustering algorithm and then the resulting clusters are used to create prototypical cases. This result of this preprocessing step will be a case base with as many or fewer cases than the input case base.

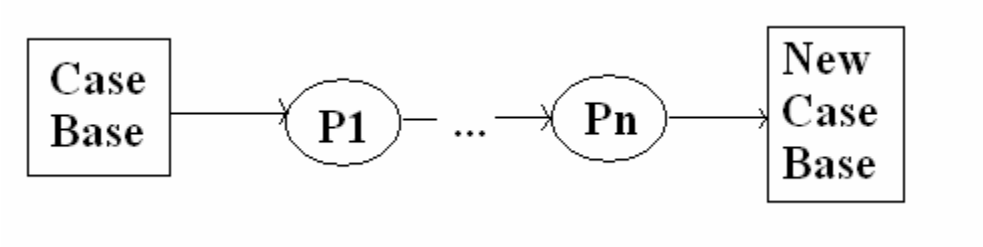


**Figure 9.12:** Preprocessing a case base through one or more steps.

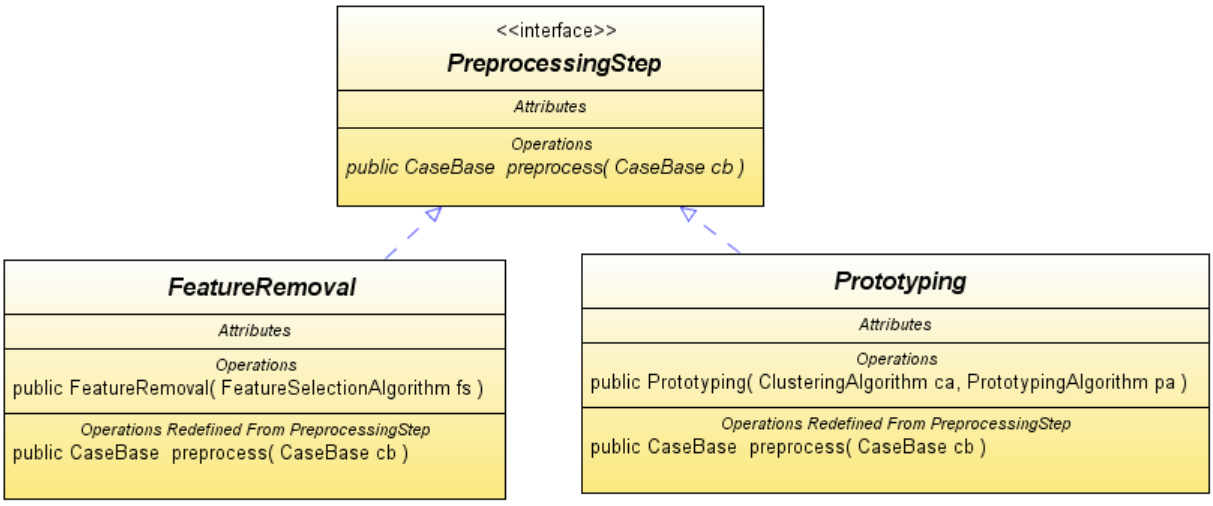


**Figure 9.13:** Class diagram of preprocessing algorithms.

Looking at Figure 9.13 we see that each preprocessing step takes in a *CaseBase*, using the *preprocess* method, and outputs a new *CaseBase*. As we can see from the figure, each implementation of a preprocessing step, feature removal and prototyping, uses the same method to perform preprocessing so the can be used interchangeably. The only difference is in the classes actually perform the preprocessing (using the algorithms supplied in their constructors).

## 9.5 Gathering Performance Metrics

The final area to consider is how the performance of a case-based imitation agent can be determined. In JIFSA, this is achieved by monitoring the behaviour of an *Agent* when presented with an input. As an example, we will examine how some of the performance metrics used in this thesis (f1, accuracy, precision and recall) are measured. In Figure 9.14 we can see the class diagram for the *StatisticsWrapper* class. This class contains the *Agent* it is used to monitor, and instead of the *senseEnvironment* method being called directly on the *Agent* it is called on the *StatisticsWrapper*. The *StatisticsWrapper* is given complete cases from an external case base, with both a problem and solution, and compares the actual case solution to the action returned by the *Agent*. As shown in Figure 9.15, the *StatisticsWrapper* first logs the action associated with an input case. It then uses the *AgentInputs* from the case as input to the *Agent* and logs the action returned. Using these expected (from the case) and actual (from the Agent) action pairs, the various performance metrics can later be computed (such as the *getGlobalF1* method).

## 9.6 Conclusion

In summary, the JIFSA framework provides several key improvements over previous case-based imitation software:

- The *Feature* class can easily be extended to introduce new features into a case.
- Each type of feature can have a unique comparison method, allowing for non-spatial features to be included in cases.
- Decoupling of the case-based imitation agent code from code used to interact with specific domains.

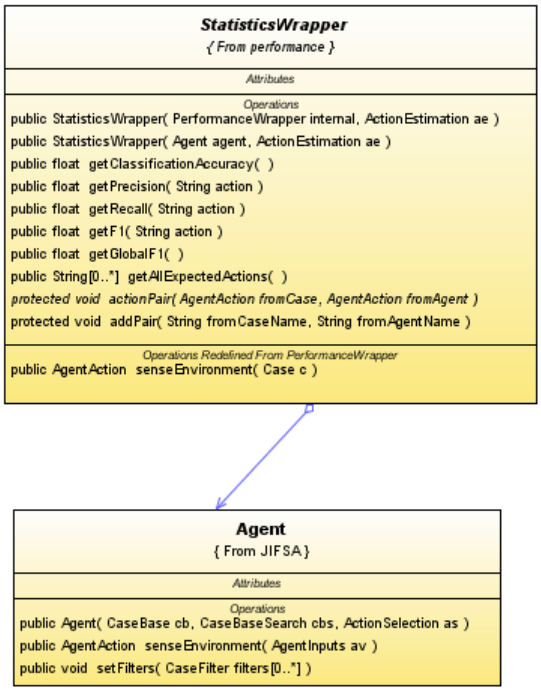


**Figure 9.14:** Class diagram for the StatisticsWrapper class.

- The ability to modify or replace any algorithm in the framework which allows for simplified expansion of the framework.
- The possibility of deploying a case-based imitation agent in a variety of environments, both simulated and physical.

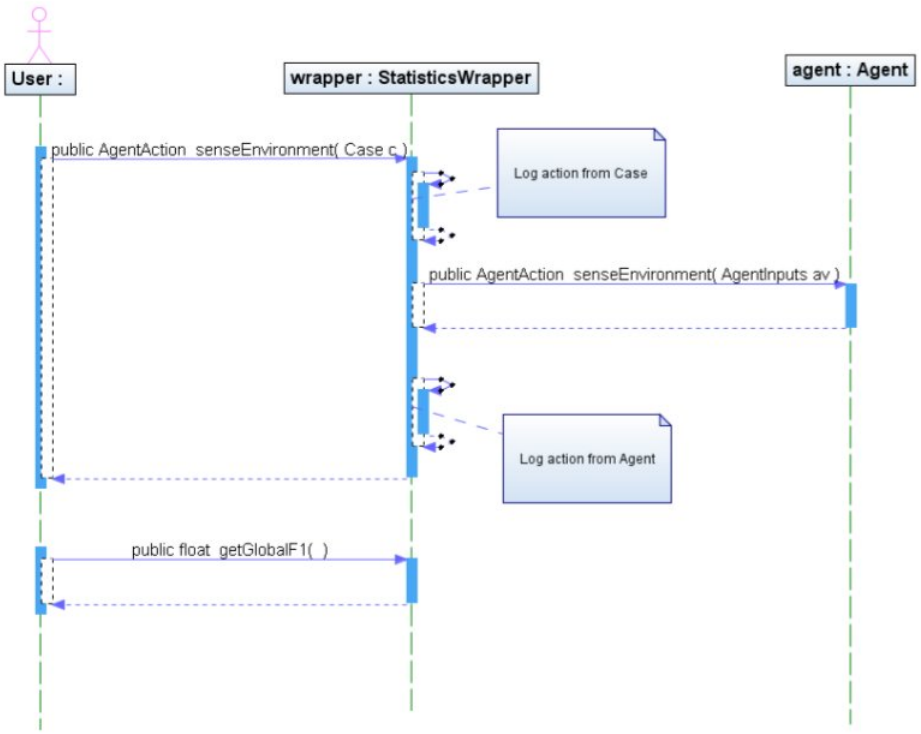


**Figure 9.15:** Sequence diagram for the StatisticsWrapper class.

# Chapter 10

# Conclusions and Future Work

This thesis has shown that preprocessing the case base used by an imitative agent, in the RoboCup simulated soccer domain, provides a significant improvement to imitative ability compared to a non-processed case base. The research question *"In the domain of simulated RoboCup soccer, how can the imitative ability of a real-time case-based imitation agent improve by preprocessing the case base it uses?"* was addressed by examining the removal of certain features from cases, clustering similar cases, creating prototypical cases from case clusters, and performing both feature removal and prototyping.

The remainder of this chapter will provide a summary for the contributions and results presented in this thesis as well as outlining the limitations of the research and future areas of work.

## 10.1 Summary of Contributions and Results

The following key contributions and results were presented in this thesis:

- **Literature review:** A description of the state of the art in imitation, real-time case-based reasoning, case-based reasoning in RoboCup soccer and case base maintenance was compiled. It was found that although imitation had

been successfully used in a variety of domains, these approaches often used extensive expert knowledge and were unable to perform a series of actions. For example, these systems could learn to both move toward a soccer ball and kick a soccer ball, but could not successfully perform those actions sequentially. While many real-time case-based reasoning applications were able to handle a series of actions, they generally also require substantial expert knowledge. The one approach which does perform a series of actions, as shown by its ability to imitate the complete behaviour of an agent, and requires minimal expert knowledge is case-based imitation of a RoboCup soccer player. The main downfall of this approach, a byproduct of reducing the expert knowledge required, is that when cases are generated in an automated manner there are often numerous redundant cases. In order to overcome this, case base maintenance operations were examined to determine if any would be applicable to the case data used by a case-based imitation agent.

- **Feature removal:** Using both binary and continuous feature selection algorithms were found to cause statistically significant increases in performance by allowing unused features to be removed from the cases. This removal allowed more cases to be searched within the real-time limits of the system, leading to the noted performance increases. These results were improved even more by dynamically setting the size of the training set used by these, based on the weight currently being tested. This approach rewarded the removal of features with larger case bases, so features with low weights were often removed in favour of the case base size increase. This approach also had the benefit of reducing the overfitting of weights, especially using continuous weighting algorithms. Thus, a binary feature weighting algorithm with a dynamically sized training set was used in subsequent chapters.

- **Case clustering:** The leader, single-linkage and distance vector clustering methods were all examined. For all metrics and teams, the distance vector approach outperformed both the leader and single-linkage algorithms, so the distance vector approach was selected to be used as the clustering method in subsequent chapters. When case bases of initial sizes of 6000, 12000, 18000 and 24000 cases were clustered into a fixed number of cluster, the consistency metrics worsened as the size of the initial case base was increased. However, the compactness and separation metrics were generally at their best values when an initial case base of size 12000 was used and then worsened for larger sized case bases.

- **Case prototyping:** Three prototyping methods were examined: using a cluster member, an average case and a range case. When the initial case base was of size 6000, using an average cluster resulted in a statistically significant performance increases over a non-prototyped case base. In many situations, but not all, the cluster member approach resulted in significant increases whereas the range case method occasional resulted in performance decreases. For larger initial case bases, the performance tended to decrease due to overgeneralization. Even for these larger initial case bases the cluster average approach tended to perform best. Thus, it was concluded the cluster average approach was the best choice for use in the RoboCup imitation domain. Also, it was found that, for an initial case base size of 6000, no significant differences occur when dirty clusters are prototyped compared to when they are not prototyped. For larger initial case base sizes, not prototyping dirty clusters provided better performance but resulted in case bases that were significantly larger than the maximum allowable size. Prototyping dirty clusters provided much more control over the final size of the case base, so that approach was used in subsequent chapters.

- **Combining feature removal and prototyping:** It was found that combining feature removal and prototyping results in statistically significant improvements compared to only using feature removal or prototyping separately. Although either ordering was found to be beneficial, performing feature removal before prototyping was found to lead to the largest increase.

- **Various teams:** The techniques were all applied to five different teams, each of which behaves in a different manner. The Sprinter, Tracker, Krislet and NoSwarm teams were all able to be successfully imitated. However, the imitator of the more complex CMUnited team still behaved noticeably different from the original. It is encouraging to note that the preprocessing techniques did significantly improve the performance of the CMUnited imitator, but more work is still required for agents of that complexity. Teams with a level of complexity that is between NoSwarm and CMUnited will need to be examined to determine what level of complexity this system can handle and guide further enhancements.

- **Software framework:** The implemented software framework provides the ability to deploy imitative software agents in a variety of domains. The framework only requires the definition of what types of sensory stimuli the agent can receive from the environment as well as the actions the agent can perform. The remainder of the learning process can be performed without any human intervention. Source code for the framework, code to use the framework in the RoboCup simulated soccer domain, data sets and videos can be found at http://rcscene.sf.net.

## 10.2 Derived Publications

The following peer-reviewed publications were derived from work or collaborations resulting from this thesis:

- M. W. Floyd, A. Davoust, and B. Esfandiari. Considerations for real-time spatially aware case-based reasoning: A case study in robotic soccer imitation. In Proceedings of the European Conference on Case-Based Reasoning (ECCBR), pages 195-209, 2008.

- M. W. Floyd, B. Esfandiari, and K. Lam. A case-based reasoning approach to imitating RoboCup players. In Proceedings of the Florida Artificial Intelligence Research Society Conference (FLAIRS), pages 251-256, 2008.

- A. Davoust, M. W. Floyd, and B. Esfandiari. Use of fuzzy histograms to model the spatial distribution of objects in case-based reasoning. In Proceedings of the Canadian Conference on AI, pages 72-83, 2008.

In addition to these written publications, work on this thesis also resulted in an award winning video. The video *"Imitating a RoboCup Soccer Player Using Case-Based Reasoning"* won *Best Educational Video* at the Artificial Intelligence Video Competition at the Association for the Advancement of Artificial Intelligence Conference 2008 (AAAI08). In addition to winning this award, it was also nominated for *Best Demonstration Video* at the same competition. Numerous professors have already indicated they have used, or are planning to use, this video as part of courses they teach.

## 10.3 Limitations and Future Work

Although this research has made improvements in the ability of an agent to imitate the behaviour of another software agent, by preprocessing the case base used by the imitative agent, there are still a number of areas that are open to further exploration. Possible future research topics relate to both overcoming the limitations of this work and addressing topics outside the scope of this thesis. Several notable areas of future work are presented:

- **Other preprocessing techniques:** This research looked to identify preprocessing techniques that could be used to improve the imitative ability of a case-based imitation agent. In some areas where existing algorithms were employed, such as feature selection and clustering, suitable algorithms were used but not necessarily ones with the highest performance or most computational efficiency. It was outside the scope of this thesis to perform an exhaustive comparison of state-of-the-art feature selection or clustering algorithms on the data.

- **Multi-state detection:** Software agents can often have multiple states of behaviour and may react differently to the same inputs depending on that state. For example, in soccer an agent may react differently depending on if it was in an offensive or a defensive state. During data collection it would be useful if the cases could be labelled according to what state the agent was in. This could ensure that the imitative agent selects cases related to a single state and does not select cases from the incorrect state at the wrong time.

- **Non-visual stimuli:** The current work only considers the visual stimuli that an agent receives, its field of vision, and does not take any other data into account. This can limit the ability of an agent to imitate a teacher, especially if

the teaching agent collaborates with other agents using inter-agent communication. In addition to communication, other non-visual data such as the time or score of a game may be useful when trying to imitate an agent. Although the implemented imitation framework can handle such non-visual stimuli, further studies will be necessary to determine the impact of including such information especially in the case of CMUnited or similarly complex teams that do make use of such stimuli.

- **Temporal link:** Cases are generated by observing a teacher over a period of time and remembering the visual inputs of the teacher and associated actions. If a temporal link was added to cases so they were aware of the case that was created at the previous moment in time, it might be possible to exploit this temporal knowledge. The cases could be considered like frames of a video, making it possible to examine the state of the environment at previous moments in time. In cluster analysis, previous temporal cases could be used as extra dimensions to differentiate between otherwise similar cases. Also, such temporal information could be used to "play back" a game to the teacher in order to see if the teacher responds similarly given the same order of input cases. If the teacher reacts differently, it could be a sign the teacher requires information not contained in the case or operates in a probabilistic manner.

- **Data acquisition:** When data acquisition occurs in an automated manner, without direct control over the teacher being imitated, it is not always possible to get the specific data that is desired. For example, when imitating a soccer agent there is no guarantee the agent will encounter a specific set of visual inputs or move to a certain area of the field. One solution would be to simulate the server and inject the desired visual inputs to the agent during the course of a soccer game. This would require gradually moving toward the desired

set of visual inputs so as to not disrupt the internal world model the teacher might maintain. For example, with a human teacher you could not simply present the teacher with random visual inputs. A human would likely not expect such significant changes in its perception especially when decisions are state-dependent.

- **Data verification:** When imitating an agent, the agent may be in a domain where it can only perform a single action per time interval. If, however, during data collection the agent performs multiple actions in a time interval it then becomes necessary to determine which action it actually meant to perform given the visual inputs. This could be performed by *playing back* cases to the agent and seeing which action the agent performs. This would essentially cause the agent to replay a game that it has previously played.

- **Other domains:** In order to create an imitation framework that is domain independent, the software framework was designed and implemented so that it had no knowledge of the environment it was being used in or what type of behaviour it would perform. Also, RoboCup agents with a variety of goals and behaviours were used as test cases. This design and experimentation, however, is no substitute for actually deploying imitation agents in multiple domains. Future work will involve imitating agents in other simulated environments, imitating physical robots and imitating humans. Not only will this test the domain independence of the software framework, but it will also open the door to new challenges such as computer vision and action recognition.

# List of References


[1] A. Aamodt and E. Plaza. Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.

[2] D. W. Aha and R. L. Bankert. A comparative evaluation of sequential feature selection algorithms. *Learning from Data: Artificial Intelligence and Statistics V*, pages 199–206, 1996.

[3] D. W. Aha and L. Breslow. Refining conversational case libraries. In *Case-Based Reasoning Research and Development, Second International Conference, ICCBR*, pages 267–278, 1997.

[4] D. W. Aha, M. Molineaux, and M. J. V. Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *International Conference on Case-Based Reasoning*, pages 5–20, 2005.

[5] M. Ahmadi, A. K. Lamjiri, M. M. Nevisi, J. Habibi, and K. Badie. Using a two-layered case-based reasoning for prediction in soccer coach. In *Proceedings of the International Conference on Machine Learning; Models, Technologies and Applications (MLMTA'03)*, pages 181–185, Las Vegas, Nevada, 2003.

[6] C. G. Atkeson and S. Schaal. Robot learning from demonstration. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 12–20, 1997.

[7] B. Auslander, S. Lee-Urban, C. Hogg, and H. Muñoz-Avila. Recognizing the enemy: Combining reinforcement learning with strategy selection using case-based reasoning. In *European Conference on Case-Based Reasoning*, pages 59–73, 2008.

[8] P. Bakker and Y. Kuniyoshi. Robot see, robot do: An overview of robot imitation. In *In AISB96 Workshop on Learning in Robots and Animals*, pages 3–11, 1996.

[9] M. J. Berry and G. Linoff. *Data Mining Techniques: For Marketing, Sales, and Customer Support*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[10] M. Bicego, V. Murino, and M. A. T. Figueiredo. Similarity-based clustering of sequences using hidden markov models. In *Machine Learning and Data Mining in Pattern Recognition*, pages 86–95, 2003.

[11] S. Chakraborti, R. Lothian, N. Wiratunga, A. Orecchioni, and S. Watt. Fast case retrieval nets for textual data. In *Advances in Case-Based Reasoning, 8th European Conference, ECCBR*, pages 400–414, 2006.

[12] 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. *RoboCup Soccer Server Manual*, 2002.

[13] A. Coates, P. Abbeel, and A. Y. Ng. Learning for control from multiple demonstrations. In *ICML '08: Proceedings of the 25th International Conference on Machine Learning*, pages 144–151, 2008.

[14] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. *INFORMS Journal on Computing*, 11(2):138–148, 1999.

[15] D. Davies and D. Bouldin. A cluster separation measure. *International Journal of Pattern Recognition and Artificial Intelligence*, 1 (2):224–227, 1979.

[16] A. Davoust, M. W. Floyd, and B. Esfandiari. Use of fuzzy histograms to model the spatial distribution of objects in case-based reasoning. In *Canadian Conference on Artificial Intellegence*, pages 72–83, 2008.

[17] K. Dorer. Extended behavior networks for the magmaFreiburg soccer team. In S. Coradeschi, T. Balch, G. Kraetzschmar, and P. Stone, editors, *RoboCup-99 Team Descriptions for the Simulation League*, pages 79–83. Linkoping University Press, Stockholm, Sweden, 1999.

[18] J. Dunn. Well separated clusters and optimal fuzzy partitions. *Journal of Cybernetics*, 4:95–104, 1974.

[19] M. W. Floyd. Software agent imitation. http://rcscene.sourceforge.net, 2008.

[20] M. W. Floyd, A. Davoust, and B. Esfandiari. Considerations for real-time spatially-aware case-based reasoning: A case study in robotic soccer imitation. In *European Conference on Case-Based Reasoning*, pages 195–209, 2008.

[21] M. W. Floyd, B. Esfandiari, and K. Lam. A case-based reasoning approach to imitating robocup players. In *Florida Artificial Intelligence Research Society Conference*, pages 251–256, 2008.

[22] D. H. Grollman and O. C. Jenkins. Dogged learning for robots. In *2007 IEEE International Conference on Robotics and Automation*, pages 2483–2488, 2007.

[23] D. H. Grollman and O. C. Jenkins. Learning robot soccer skills from demonstration. In *IEEE International Conference on Development and Learning*, 2007.

[24] J. A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 1975.

[25] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, August 2001.

[26] G. H. John, R. Kohavi, and K. Pfleger. Irrelevant features and the subset selection problem. In *International Conference on Machine Learning*, pages 121–129, 1994.

[27] A. Karol, B. Nebel, C. Stanton, and M.-A. Williams. Case based game play in the robocup four-legged league part i the theoretical model. In *RoboCup 2003: Robot Soccer World Cup VII*, pages 739–747, 2003.

[28] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.

[29] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[30] D. Krznaric and C. Levcopoulos. Fast algorithms for complete linkage clustering. *Discrete and Computational Geometry*, 19(1):131–145, 1998.

[31] K. Lam. A scene learning and recognition framework for robocup clients. Master's thesis, Carleton University, Ottawa, Canada, 2005.

[32] K. Lam, B. Esfandiari, and D. Tudino. A scene-based imitation framework for robocup clients. In *MOO - Modeling Other Agents from Observations*, 2006.

[33] K. Langner. The Krislet Java Client. http://www.ida.liu.se/ frehe/RoboCup/Libs/libsv5xx.html, 1999.

[34] D. B. Leake and D. C. Wilson. Categorizing case-base maintenance: Dimensions and directions. In *Advances in Case-Based Reasoning, 4th European Workshop, EWCBR*, pages 196–207, 1998.

[35] D. B. Leake and D. C. Wilson. Remembering why to remember: Performance-guided case-base maintenance. In *Advances in Case-Based Reasoning, 5th European Workshop, EWCBR 2000*, pages 161–172, 2000.

[36] C. Marling, M. Tomko, M. Gillen, D. Alexander, and D. Chelberg. Case-based reasoning for planning and world modeling in the robocup small sized league. In *IJCAI Workshop on Issues in Designing Physical Agents for Dynamic Real-Time Environments*, 2003.

[37] P. Marlow. A process and tool-set for the development of an interface agent for use in the RoboCup environment. Master's thesis, Carleton University, 2004.

[38] M. Molineaux, D. W. Aha, and P. Moore. Learning continuous action models in a real-time strategy environment. In *Florida Artificial Intelligence Research Society Conference*, pages 257–262, 2008.

[39] J. M. Pe na, J. A. Lozano, and P. Larra naga. An empirical comparison of four initialization methods for the k-means algorithm. *Pattern Recogn. Letters*, 20(10):1027–1040, 1999.

[40] K. Racine and Q. Yang. Maintaining unstructured case bases. In *Case-Based Reasoning Research and Development, Second International Conference, ICCBR*, pages 553–564, 1997.

[41] J. Reunanen. Overfitting in making comparisons between variable selection methods. *Journal of Machine Learning Research*, 3:1371–1382, 2003.

[42] M. A. Riedmiller, A. Merke, D. Meier, A. Hoffman, A. Sinner, O. Thate, and R. Ehrmann. Karlsruhe Brainstormers - a reinforcement learning approach to robotic soccer. In *RoboCup 2000: Robot Soccer World Cup IV*, pages 367–372, London, UK, 2001. Springer-Verlag.

[43] RoboCup. Robocup official site. http://www.robocup.org, 2008.

[44] H. Romdhane and L. Lamontagne. Reinforcement of local pattern cases for playing tetris. In *Florida Artificial Intelligence Research Society Conference*, pages 263–268, 2008.

[45] R. Ros and J. L. Arcos. Acquiring a robust case base for the robot soccer domain. In M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 1029–1034. AAAI Press, 2007.

[46] R. Ros, R. L. de Mntaras, J. L. Arcos, and M. Veloso. Team playing behavior in robot soccer: A case-based approach. In *ICCBR 2007, 7th International Conference on Case-Based Reasoning*, Lecture Notes in Artificial Intelligence, pages 46–60. Springer, 2007.

[47] R. Ros and M. Veloso. Executing multi-robot cases through a single coordinator. In *Proceedings of AAMAS'07, the Sixth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Honolulu, Hawaii, May 2007.

[48] R. Ros, M. Veloso, R. L. de Mntaras, C. Sierra, and J. L. Arcos. Retrieving and reusing game plays for robot soccer. In *Advances in Case-Based Reasoning. 8th European Conference on Case-Based Reasoning (ECCBR-06), Fethiye, Turkey, September 4-7, 2006*, volume 4106 of *Lecture Notes in Artificial Intelligence*, pages 47–61. Springer, 2006.

[49] A. Sarje, A. Chawre, and S. B. Nair. Reinforcement learning of player agents in RoboCup soccer simulation. In *Proceedings of the Fourth International Conference on Hybrid Intelligent Systems (HIS'04)*, pages 480–481, Washington, DC, USA, 2004. IEEE Computer Society.

[50] B. Smyth and M. T. Keane. Remembering to forget: A competence-preserving case deletion policy for case-based reasoning systems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI*, pages 377–383, 1995.

[51] B. Smyth and E. McKenna. Building compact competent case-bases. In *Case-Based Reasoning and Development, Third International Conference, ICCBR*, pages 329–342, 1999.

[52] T. Steffens. Adapting similarity measures to agent types in opponent modeling. In *Proceedings of the Workshop "Modelling Other Agents from Observations" at AAMAS 2004*, pages 125–128, 2004.

[53] P. Stone, P. Riley, and M. M. Veloso. The CMUnited-99 champion simulator team. In M. M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup*, volume 1856 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 1999.

[54] N. Sugandh, S. Ontañón, and A. Ram. On-line case-based plan adaptation for real-time strategy games. In *Association for the Advancement of Artificial Intelligence Conference*, pages 702–707, 2008.

[55] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, May 2005.

[56] C. Thurau and C. Bauckhage. Combining self organizing maps and multilayer perceptrons to learn bot-behavior for a commercial game. In *In Proceedings of the GAME-ON Conference*, 2003.

[57] J. Wendler and M. Lenz. Cbr for dynamic situation assessment in an agent-oriented setting. In *AAAI-98 Workshop on Case-Based Reasoning Integrations*, 1998.

[58] D. Wettschereck and D. W. Aha. Weighting features. In *ICCBR '95: Proceedings of the First International Conference on Case-Based Reasoning Research and Development*, pages 347–358, London, UK, 1995. Springer-Verlag.

[59] D. C. Wilson and D. B. Leake. Maintaining cased-based reasoners: Dimensions and directions. *Computational Intelligence*, 17(2):196–213, 2001.

[60] R. Xu and D. I. I. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

[61] Z. Zhang and Q. Yang. Towards lifetime maintenance of case based indexes for continual case based reasoning. In *Artificial Intelligence: Methodology, Systems, and Applications, 8th International Conference, AIMSA*, pages 489–500, 1998.

[62] Y. Zhao and G. Karypis. Empirical and theoretical comparisons of selected criterion functions for document clustering. *Machine Learning*, 55(3):311–331, 2004.

[63] J. Zhu and Q. Yang. Remembering to add: Competence-preserving case-addition policies for case base maintenance. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI*, pages 234–241, 1999.