INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

A Scene Learning and Recognition Framework for RoboCup Clients

by

Kevin Lam, B. Eng, Carleton University

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 in Electrical Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
August 2005

The undersigned recommend to the

Faculty of Graduate Studies and Research

acceptance of the thesis

A Scene Learning and Recognition Framework for RoboCup Clients

Submitted by Kevin Lam, B. Eng, Carleton University

in partial fulfillment of the requirements for

the degree of Master of Applied Science in Electrical Engineering

Thesis Supervisor
Dr. Babak Esfandiari

Chair, Department of Systems and Computer Engineering
Dr. Rafik A. Goubran

Abstract

As multi-agent systems grow in complexity and diversity, they become increasingly difficult to design. Agents are described in terms of their behaviour, typically trained by an expert who prepares knowledge representations or training data for supervised machine learning.

To reduce development time, agents could learn by observing the behaviour of other agents. This thesis describes an effort to train a RoboCup soccer agent by capturing data from existing players, generating a knowledge representation, and using a real-time scene recognition system. The trained agent later exhibits behaviour traits similar to the observed agent and can appear to completely imitate the behaviour of the original; the process requires little human intervention.

Experiments are performed using three agents of varying complexity. The "scene" knowledge description format, and simple scene matching algorithm, are limited to imitation of stateless and deterministic agent behaviours. Future work includes improving the matching algorithm and developing higher-level behaviour models.

Acknowledgments

Many thanks go to Professor Babak Esfandiari for his continued support, guidance and supervision during every phase of research, design and experimentation, and while this thesis was being written.

Thanks also go to Paul Marlow, for his help and feedback in many areas, particularly including his development and continued support of several tools used for the experiments herein.

The author also wishes to acknowledge the contributions of the following Carleton University students who participated in the development, testing and enhancement of the various algorithms described herein:

Khurrum Ashfaque	Saurabh Bagrodia	Laurentiu Checiu	Deryck Velasquez
Shawn Henry	Chris Kafka	Ding Wei	Fan Ping Zhou
Benoit Essiambre	David Tudino	Chris Desmarais	Alan Wai
Rachid Chreyh	James Kelly		

Finally, many thanks go to family and friends for their continuing and gracious support and encouragement, without whom this effort would not have been possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

> To do successful research, you don't need to know everything, you just need to know of one thing that isn't known.
>
> *Arthur Schawlow*

1.1 Motivation

In recent years, a new paradigm in computing has been gaining in interest and popularity — software agents [54, 77]. Generally such agents operate autonomously and are situated in a particular environment, interacting with their environment, particularly with human users or with other software agents [77]. As multi-agent systems grow in complexity and diversity, they become increasingly difficult to design. It is impractical for agent designers to attempt to hard-code all possible interaction situations into the software [67]. Instead, it becomes much more useful to describe agents in terms of their "behaviour" [67, 86].

Autonomous software agents are typically deployed to perform tasks on behalf of,

or instead of, a human user. In a collaborative (or competitive) environment where agents communicate and interact with each other, the action an agent takes is related to both the ground rules directing the agent's ultimate goal as well as the specific interplay between the decision-making agent and all the entities in its surroundings.

An agent's behaviour may be described by its reaction to particular situations in the environment — either temporal or spatial. An agent may react to a sequence of inputs over a period of time [18, 22], or a particular spatial configuration of entities around it [57, 67], or a combination of both. It is these types of "scenarios" [86] that describe the interesting behaviours of the agent.

A successful and effective software agent must develop an accurate, relevant knowledge representation of its domain environment, complementing its decision-making facilities. Typically, this knowledge representation is generated by a human domain expert, who either hard-codes the data or algorithm or facilitates a machine learning process by filtering or fine-tuning the input data. Both approaches can be very time-consuming, and if the strategies to be implemented or learned are complex, this can involve a lot of effort on the trainers behalf. An ideal approach should produce effective and accurate behaviour, with minimal time and effort spent on training the agent.

1.2 Learning by Imitation

In certain domains it may be desirable to recognize or learn behaviour directly from another agent or other entity; for example, when trying to predict the strategy of an adversary (e.g. in a game environment, or bidding in an e-commerce system). In a competitive environment, it may be highly beneficial for an agent to be able to observe

the behaviour of its competitor, whether in order to counteract the competitor's behaviour, or to emulate it. Such an approach may be used to learn only specific skills [70, 71] or to characterize more higher-level behaviour.

Ideally, the agent should learn as much as possible with minimum effort by the domain expert. Consider a human who seeks to learn from an expert (a master and apprentice relationship). While some of the interaction between the two is explicit training, the apprentice learns much simply by observing and imitating the master. The goal can then be stated as a desire for an agent to learn to imitate the behaviour of a domain expert. Indeed, the goal of most machine learning is to infer a computation that imitates the rule that generated the input data.

In general terms, the observed agent is treated as a "black box" which processes inputs and generates an output in response. Consider the black box as an implementation of some function f such that

$$y = f(x_1, x_2, \cdots, x_n) \tag{1.1}$$

where y is the output decision for any given circumstance involving input attributes $x_1, x_2, \cdots, x_n$. The goal of agent imitation is to discover an approximation for f, given a training set P of points $p(y, x_1, x_2, \cdots, x_n)$.

Imitative behaviour, particularly for learning repetitive tasks, is the staple of many interface agents, particularly those that monitor human input to a computer system. The domain expert need not even be human; it may be another agent, with more experience or training.

1.3 The RoboCup Soccer Environment

Consider, as a domain environment for autonomous agents, the RoboCup (Robot World Cup) Simulation League [24]. The Simulation League provides a testbed for the design of autonomous software agents given the specific domain of a game of soccer.

In the RoboCup Simulation League, agents represent individual players on a soccer team. As such, agents are autonomous, situated in a spatial environment. The usual objective for a RoboCup soccer team is to score goals by kicking the ball into the opponents goal; agents may attempt to realize this goal individually or, more commonly, collaborate with teammates by passing the ball or implementing other high level strategies such as goaltending, defense or offensive plays. Agents must work to realize their goal while also attempting to prevent the agents of the opposing team from achieving theirs.

RoboCup provides a real-time, dynamic and non-deterministic environment which is similar in nature to many other real-world environments (such as communications networks, the Internet, vision, statistical analysis) and a common platform over which various agents can be compared and evaluated. A wide variety of teams and researchers are actively developing clients for RoboCup, introducing new algorithms and improving on known ones. Thus, there is a wide body of existing research to draw from, and, for data mining approaches, large bodies of existing raw data available. These characteristics make it an ideal platform for agent research.

1.4 Objectives

The goal of this thesis is to address the problem of requiring large amounts of time and human effort to train an agent in the RoboCup domain. This research was built upon an initial project to develop a human interface agent for RoboCup, in order to train a software agent from the actions of a human player through a GUI interface [51]. That project is ongoing, though preliminary results show that while humans may be capable of playing more intelligently than software entities, the lack of a robust GUI interface and the speed of the software agents made it difficult for a person to play alongside (see also [66]).

This thesis instead describes an attempt to develop a process by which a RoboCup soccer agent can learn its higher-level algorithm (how to play soccer) by observing another agent and imitating its behaviour. Instead of human players, other existing RoboCup teams are observed. Since there is a known set of competent teams from the annual Robot World Cup games, it should ideally be possible to learn from, and subsequently emulate, existing teams. These players act as "coaches" from which the agent can observe and learn; consequently the trained agent, when placed in a similar environment, should display some degree of behaviour similarity. The algorithms must be robust, requiring as little human intervention as possible, so the process may be fully automated.

Because of the spatial nature of the RoboCup environment, our methodology involves observation and capture of sensory data from RoboCup agents, developing a data representation revolving around "scenes" — snapshots of the environment at given points in time — and develop a RoboCup client agent around an appropriate scene recognition algorithm.

1.5 Contributions

First, we provide an overview of the current state in agent imitation through observations, and provide a summary consideration of the factors, considerations and limitations involved in imitative learning.

We have developed an extensible framework for research in RoboCup agent imitation, which includes the following components:

- a conversion process for transforming raw RoboCup player logs into a higher-level knowledge representation format
- a set of algorithms to support scene recognition and matching using a customizable k-nearest-neighbor approach
- a RoboCup client agent based on this scene recognition algorithm

We provide results which demonstrate an agent's ability to imitate the high-level behaviour of another agent, without the use of extra training, algorithm tweaking, or reinforcement learning applied. We show the ability to present these results in a semi-automated fashion requiring very little human intervention. These results demonstrate that it is possible to imitate the behaviour of simple spatial agents in RoboCup using a simple recognition algorithm.

However, the current approach is certainly not without limitations. The current scene-imitation agent is itself stateless and deterministic, which suggests that it would have difficulty representing any other agent which is not so (this is indeed shown to be the case). This imposes limits on the extent of which the current algorithm can imitate other agent behaviours. The current algorithms are also not optimized for

space or time. Suggestions for future work are discussed based on these limitations and ideas for improvements.

We also show the results of using this simple scene recognition engine as a basis for a term project in a graduate course in software agents, and discuss contributions made by the student projects. Finally, we provide analysis on potential future enhancements to the scene recognition system with the intent of overcoming the current limitations.

1.6 Thesis Structure

The RoboCup soccer simulation league was chosen for its use of real-time, spatially situated agents in a highly tangible environment, its easy availability of training data, and its wide community of agents (players) and the researchers developing them. Chapter 2 motivates the use of RoboCup and provides more detail about the RoboCup domain.

Chapter 3 considers the current state of research in the related problem domains, including that of RoboCup agent development and typical approaches, interface agents, agent modelling through observation, spatial and scene recognition, pattern recognition and other learning techniques.

Chapter 4 describes the overall methodology, including detailed requirements and other considerations. Chapter 5 describes the development of an appropriate scene description format, while Chapter 6 describes the problem of scene recognition. We provide insights into the general problem of scene recognition and present several algorithms for use with the scene description format. Chapter 7 describes the implementation process from capturing data from existing RoboCup clients to building a RoboCup client based on the scene recognition framework.

Chapter 8 shows the results, both qualitative and quantitative, from the RoboCup agent when placed "in the field". We consider the results of attempting to imitate each of three different RoboCup teams, using the different scene algorithms and parameters presented previously.

Finally, Chapter 9 presents the conclusions and contributions made. We examine the limitations of the curent process and present possible ways to overcome them in future work.

Chapter 2

RoboCup Overview

> That's what learning is, after all; not whether we lose the game, but how we lose and how we've changed because of it and what we take away from it that we never had before, to apply to other games. Losing, in a curious way, is winning.
>
> *Richard Bach*
> THE BRIDGE ACROSS FOREVER

The following sections provide a detailed overview of the Robot World Cup Initiative (RoboCup) including an introductory overview and brief history, a description of the RoboCup simulation league, and a technical overview of the communications protocol used by RoboCup software agents.

2.1 RoboCup Overview

The Robot World Cup Initiative, or RoboCup, is an attempt to foster AI and intelligent robotics research [24] by providing a standard problem environment, through which a variety of technologies may be developed, combined and tested. Just as chess

is a standard problem for demonstrating an AI's "intelligence", RoboCup provides a platform through which a variety of strategies, algorithms and techniques may be compared and empirically evaluated.

Annual "World Cup" championships ensure that RoboCup-based research is motivated and performed in a spirit of light-hearted competition, while workshops and conferences provide an arena for academic and research discussion. Current research topics include the design of autonomous software agents, multi-agent collaboration, reasoning and strategy, sensor analysis, real-time visual data acquisition, and real-time deployment [36, 80].

Perhaps the most ambitious goal of the RoboCup initiative is to "by the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team." [24] While research may yet be ongoing in the humanoid RoboCup league, there are also active communities for four-legged robots (i.e. Sony AIBO dogs), middle- and small-sized robot leagues, and a RoboCup simulation league. The latter allows for research and development of RoboCup client agents without requiring any physical hardware.

2.2 The RoboCup Simulation League

The RoboCup simulation league is an ideal environment for research in autonomous agents because of its ability to provide a single, common platform over which agents can be compared and evaluated. The overarching "problem statement" is simple — develop a team of agents which work together to play a simulated game of soccer — yet its deployment may take many forms (Section 3.1 describes a number of popular approaches). Through competition games, different approaches can be compared to

determine their relative effectiveness. The simulation league is purely software, with games displayed on a monitor (Figure 2.1).

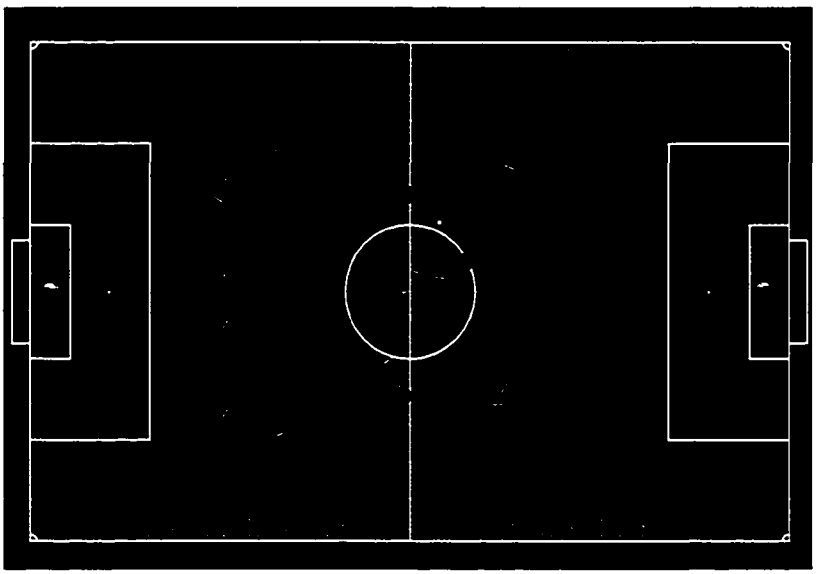


Figure 2.1: A typical game screenshot from the RoboCup simulation league.

RoboCup, by its nature, is a real-time, dynamic and non-deterministic environment. As in real-world soccer, the data available to players (client agents) is somewhat uncertain, i.e. noisy. Most real-world environments (for example, communications networks, the Internet, machine vision, statistical analysis) tend to also exhibit similar characteristics. Thus, research in the RoboCup simulation environment may also have a high degree of relevance in these other application domains.

Client agents in RoboCup must deal with temporal events as well as with an environment consisting of a two-dimensional space (the soccer field), objects within the space and other agents, both protagonistic and antagonistic, situated within the space. Because the simulator only models a two-dimensional world, game physics are simplified (ball travel, for example, has no height component). A three-dimensional

world model, with corresponding simulator, clients, and a corresponding 3-D visualization system, are currently under development.

The simulation league [29, 10] spans multiple computing platforms and operating systems and supports any programming language capable of handling UDP socket communications. Although tools exist to display the progress of a simulation soccer game graphically, they are not required; hence, simulations can be run in unattended batches or on relatively low-power hardware where real-time performance is not required.

2.3 RoboCup Communications Protocol

The RoboCup simulator is based on a client/server architecture, with a single RoboCup soccer server [29] and multiple clients for the players. The central soccer simulator server tracks game progress and all game physics over the duration of the game; each player is represented by a single client, with no communication allowed between clients. All communications are standard UDP packets so the clients and server can be run on multiple machines over a network. A typical team consists of up to 11 player clients (one of which may be designated as the goalie), and an optional coach client.

The soccer game occurs over a discrete time period (a standard game is 6000 simulator time cycles). During each cycle, the server provides every client with information regarding their state and world view, including what objects are currently in proximity, their distances and angles, as well as the current velocity of the player, orientation of its head, and other key physical information. Inputs correspond to visual, aural, and body awareness. All data is subject to a noise model based on distance

— the farther away an object is, the less certain is its distance and description. The clients then send a command to the server in response, typically to perform an action such as a dash, kick, or turn. Figure 2.2 illustrates the data exchange during a time cycle.

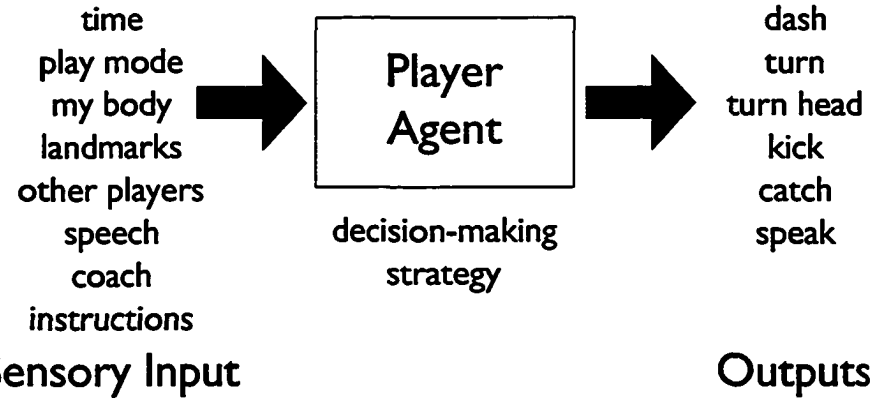


Figure 2.2: RoboCup clients receive sensory inputs, generate a decision, and send a command to the server.

An additional class of RoboCup clients was recently introduced, known as the RoboCup coach client [10]. Coach clients (e.g. [58, 38]) are privileged in that they can receive noise-free data about the entire soccer field, but they are restricted to only sending occasional communications to teammates. Their intended purpose is not to "micromanage" their teams but to provide strategic decision-making abilities.

Finally, monitor clients [29, 12] are used to visualize the action during a game, as well as provide a means for human input (for example to pause the game, or to determine kickoff times). Any number of monitors may connect with the server simultaneously. A monitor typically provides a top-view 2-dimensional overview of the game, though specialized monitors have been developed for other purposes.

2.3.1 Message Formats

In a typical exchange, the client connects to the server on a known UDP port (6000 by default). After an initial handshaking protocol, the client and server begin sending messages to each other on a newly allocated port. Messages follow the general format:

```
(message_type [params])
```

The following sections describe the types of messages that are typically exchanged between client and server during the course of a game.

2.3.2 Server Messages

Messages from the server generally concern the state of the client and the objects around it, which the client may interpret as it sees fit. There are three types of state messages (Table 2.1), each subject to a model of noise and uncertainty.

Table 2.1: RoboCup server status messages

Message type	Format
sense_body	(sense_body Time (view mode) (stamina etc. $\cdots$)
see	(see Time ObjInfo ObjInfo $\cdots$)
hear	(hear Time Sender "Message")

Objects described in the see message may be players, the ball, goals, or the numerous lines and flags located on the field (Figure 2.3). Objects described by the server include attributes such as distance, direction, and deltas for distance and direction if the object is in motion. When describing soccer players, the direction the

body and head are facing are also given, if available, as well as the player's team and uniform number.

Hear messages may originate from players who say things (including the speaking agent itself), or from the game "referee". The server maintains a state machine of the current game mode (kick off, game on, goals, penalties, etc.) and announces scoring and state changes through these messages.

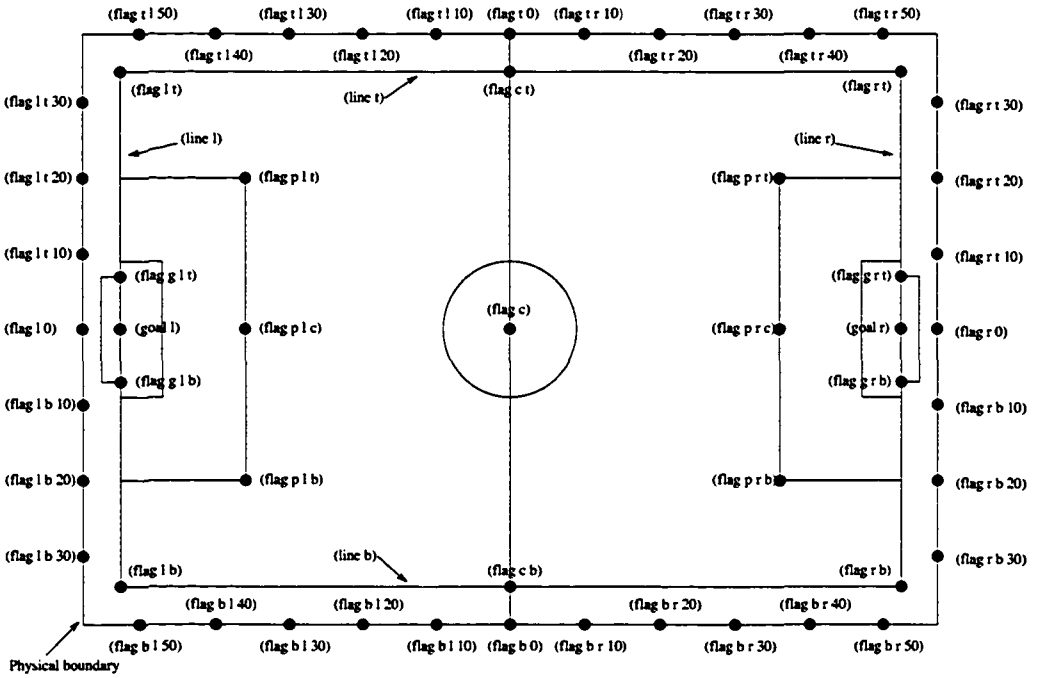


Figure 2.3: RoboCup field lines and flags (from the RoboCup Soccer Manual [10]).

2.3.3 Player Commands

RoboCup clients are responsible for interpreting the received messages and responding in turn with action commands (Figure 2.2). Only one "intentional" action (an action that impacts the agents' surroundings) may be sent in any given time cycle. Clients

may say messages as frequently as desired, though, as in the real world, messages can only be heard for a limited distance, and delivery is not guaranteed (particularly if many say messages are sent simultaneously). It is the player's responsibility to interpret the semantic meaning of any utterances sent from its teammates, as these are generally proprietary from team to team.

All communications occur asynchronously.

Table 2.2: RoboCup player client commands

Client Command	Once/Cycle	Notes
(catch Direction)	Yes	Goalies only
(change_view Width Quality)	No	More detail vs. wider view
(dash Power)	Yes	
(kick Power Direction)	Yes	
(move X Y)	Yes	Only during before_kick_off
(say Message)	No	
(sense_body)	No	resend sense_body message
(score)	No	resend current score
(turn Moment)	Yes	
(turn_neck Angle)	Yes	

2.4 RoboCup Motivation

RoboCup, and the Simulation League in particular, is a well-established, popular domain environment for software agent research in a number of areas. It is an ideal domain for research in imitative agents, as it provides the following key advantages:

- a dynamic, non-deterministic environment suitable for real-world modelling and applicable to many other domains;
- cross-platform availability of simulator binaries;

- availability of platform- and programming language-agnostic tools, including log analyzers, converters and learning tools;
- a well-defined set of game rules and communication protocol;
- a large existing research community actively developing tools, agent teams and the simulation server;
- a wide body of existing research to draw from, and easily-obtainable data for data mining;
- existing client frameworks for rapid deployment of new RoboCup player agents;
- a tangible means of evaluating agent performance by considering in-game performance.

This combination of characteristics makes the RoboCup Simulation League an ideal platform for software agent research.

Chapter 3

State of the Art

> Keep on the lookout for novel ideas that others have used successfully. Your idea has to be original only in its adaptation to the problem you're working on.
>
> *Thomas Edison (1847-1931)*

This chapter provides an overview of several different areas of research that pertain to the goal of creating a RoboCup Simulation League team. Given our goal of developing a RoboCup team that observes and imitates behaviour from other players, our intent is to wherever possible reuse agent architectures, knowledge formats, algorithms, and tools. To support this goal, we consider:

- Existing RoboCup client frameworks (architecture and approaches)
- Knowledge representation formats (including spatial knowledge representation)
- Agent observation and modeling
- Scene recognition and scene-matching approaches
- Learning from direct observation (imitative learning)

In addition to providing guidance on potential reuse of existing approaches (or following popular trends observed in current approaches), the chapter concludes by motivating the use of imitative learning in the context of developing and training a RoboCup agent.

3.1 RoboCup Client Frameworks

A large number of RoboCup client teams have been developed since the inception of the simulation league in 1997, most from academic institutions around the world (e.g. [68, 69, 80, 36, 16, 46]). They span a variety of platforms, languages, algorithms and capabilities. Some of the more prominent clients (such as CMUnited [69, 71, 70]) have formed the basis for development of newer clients (e.g. Dirty Dozen, TsinguAeolus, FC Portugal [58, 60, 82]) each improving on the design or performance of their predecessors.

There are several common framework types into which most existing RoboCup clients can be categorized:

- simple clients (hard-coded or scripted)
- basic learning clients (single- and multiple-layers)
- player-influenced learning clients

The following sections describe these techniques and the motivation behind them, citing examples representative of each.

3.1.1 Simple Clients

The most straightforward implementation of a RoboCup agent is simply to hard-code or script its desired behaviour. The extremely simple clients are simply hard-coded, and any behavioural changes must be reprogrammed. However, the limitations of purely hard-coded approaches (difficulty in modifying and adapting strategy, difficult maintenance, limited scalability, impractical amount of code required to model a complex domain environment, and the need for a domain expert who is also a programmer) conflict with RoboCup's goal of autonomous, intelligent player behaviour. Most hard-coded RoboCup players are designed as tests, demonstrations, prototypes or starting points for further development: Krislet [40] is a prototype of a development framework, and NewKrislet [32] is used primarily in a curriculum for teaching programming to students.

One step up from basic hard-coding of behaviour is the separation of knowledge from the rest of the code. Programmed algorithms are separated from the behavioural rules guiding the agent, allowing the agent's strategy to be modified without affecting the code. Decision rules or state machines, for example, may be defined in a a user-editable file.

Many agents use some variation of a state machine. The agent's code defines all possible states and transition functions. An external, user-editable file encodes the actual state machine (i.e. transitions between states). Files can be written in a proprietary format, human-editable ASCII, or XML. Some clients also support numerical parameters such as distances and thresholds.

The simple RoboPlayer client [41], for example uses a finite state machine logic implementation, reading in player configuration files described in XML. magmaFreiburg

[16] and FC Portugal [60] are two examples of RoboCup clients that use an external representation for large portions of the player behaviours. magmaFreiburg (which implements a state-machine-like extended behaviour network) sports a GUI interface which allows a human supervisor to load or modify the behaviour description files; sample files are provided to define separate behaviours for defenders, offenders, midfield and goalie players. FC Portugal allows the human coach to select strategies and suggest formations and tactics both before and during the game. The players make their own decisions on ball passing, kicking, etc. by evaluating metrics specifying distances, confidences and expected probabilities.

Case Study: Krislet

Krystof Langner's *Krislet* [40] is an example of a very simple hard-coded soccer agent written in Java. Despite its extreme simplicity, Krislet's performance is surprisingly effective, even against far more complex agents, largely because of the simple, fast nature in which it operates.

Krislet follows a simple set of decision rules, which state the following:

1. If the ball is not within sight, turn to the right by 40 degrees.
2. If the (visible) ball is not within 1 meter from the player, run toward the ball.
3. If the ball is within 1 meter, and the opponent goal is not within sight, turn to the right by 40 degrees.
4. If the ball is within 1 meter, and the opponent goal is in sight, kick the ball toward the goal.

Krislet's rules (essentially a decision tree) are hard-coded as if/else statements.

Krislet's simple yet effective strategy demonstrates the potential merit of using low-level rule generation, showing that a simple approach (even a hard-coded one) can be viable.[1]

Summary

Table 3.1 lists a few common methods and examples of simple RoboCup clients, including hard-coded and scripted agents.

Table 3.1: Some methods for creating simple RoboCup clients, with examples.

Method	Implementations	Examples
Decision Trees	Hard-coded if/then statements, tree structures, WEKA	Krislet [40], Zed [79]
State Machines	Programmed state machines, Hidden Markov Models, Push-Down Automaton, Extended Behaviour Network	NewKrislet [32], RoboPlayer [41], magmaFreiburg [16], FC Portugal [60]
Goal-Seeking	STRIPS planning, BDI	RoboLog [68], Stripslet [56], also [46, 83]

The separation of knowledge from implementation introduces a level of abstraction away from the basic functions of a RoboCup player. A hard-coded library of skills (such as run, kick, turn) can be provided along with basic logical predicates (*haveBall*, *BallInArea*) — it is up to the domain expert (who no longer needs to necessarily be a programmer) to connect these together to describe the desired behaviour. Multiple behaviour descriptions can coexist and may be used to provide a distinction between

[1] An entire (unrelated) field of robotics research, BEAM robotics, considers the use of hardwired rules in robotics, seeking to build robots that display complex behaviors with little computing power. [48] Such minimalist strategies are frequently compared to biological forms and are cited as being insect-like. A consequence of their simplicity is that such robots have no way to learn from past behaviour or be programmed to perform useful sequential tasks.

different player types (such as the goalie versus an defender); the agent's core logic need not change.

This functional separation has clear benefits:

- it removes the need for a programmer when the behaviour needs to be changed
- different agents using the same base code can each have different behaviours (e.g. forwards vs. goalies)
- the configuration file is (usually) easier to understand than program code

Though this model introuces much more flexibility, the need for a domain expert is still very strong; someone must still encode the instructions or state machines. Fine-tuning or debugging the encoding is still a trial-and-error process. However, the use of a formally defined representation suggests the possibility of using machine-generated knowledge, whereby rules (or parameters) are derived automatically from the output of a machine learning process.

3.1.2 Machine Learning Approaches

The state-driven designs seen previously are largely designed by human domain experts. Parameters may also be fine-tuned (for example, to specify the distance away from the ball before a logical predicate such as *have-ball* becomes true). These two design problems are candidates for applying learning techniques:

- using learning techniques to fine-tune low-level skills
- using learning techniques to build the high-level decision rules

Learning Skills

Reinforcement learning algorithms for fine-tuning skills are common, particularly in solving the ball-passing problem. Agents can learn when to pass to teammates (e.g. learning the "passability" of different situations [9, 8] and judging when and where to pass) or learn to intercept passing attempts by opposing teams [83, 84]. Other potential learning areas include aiming and dribbling [70], and self-localization on the field (e.g. [26, 85] describe a feed-forward neural network to localize a physical robot in the RoboCup Legged Robot League).

Common methods for low-level skill learning include using decision trees, neural networks, Q-Learning [83], and fuzzy inference systems [64, 63]. Generally these are supervised learning techniques; the training data for these algorithms are typically hand-picked from game logs or specifically manufactured to cleanly demonstrate positive and negative training instances. Special environments or tools may be required to create the training set. Additionally, these solutions typically are only applicable to training single agents, and do not scale well to multi-agent interaction [64].

Learning Rules

When using a learning process to learn high-level game rules, the goal is to develop a mapping between the agent's sensors (world view inputs from the RoboCup server) and its actuators (its impact on the environment) [71]. What was previously scripted by a domain expert must now be trained and generated through a learning process. This typically requires a large amount of training data (such as the raw observed logs from RoboCup games [34, 38]); it may also require "sanitizing" of some data to assist the learning process [19]. Learned rules are typically represented as decision trees,

which are easy to understand and to validate.

Common methods include decision-tree induction (both [19] and [38] use this method to determine rules which are then fed to a RoboCup coach client; each coach then uses these rules to advise its team), and the use of inductive logic programming (where a set of first-order-logic predicates are made available). Matsui et al [52] and Jacobs et al [34] describe ILP logic implementations which scan the training data and produce decision tree rules. The latter uses the rules in an offline validation and verification process rather than applying them directly to an agent's behaviour.

Case Study: ILP Rule Generation

Matsui et al [52] describe an "inductive learning agent" which uses inductive logic programming to learn rules. Examples of actions are collected and classified; ILP is applied to generate the rules which the agent then follows. The agent can then predict the success or failure of rules and adjust its behaviour accordingly. The CMUnited agent [69] is used as a basis for the inductive learning agent.

The ILP agent makes use of defined predicates such as *ball_distance*, *ball_angle_from_body*, *player_speed*, to describe the agent state and environment, and a defined list of actions such as *kick_to_goal*, *receive_ball*, *pass_ball*, etc.

Experiments were conducted giving the agent goals such as to shoot or to pass (knowledge is represented as a set of desired goals, from which the ILP system deduces the rule set necessary to succeed in accomplishing the goals). The rate of success was 36.2% for the shooter agent and 31.9% for the passer agent. The agent also derived rules to predict the outcome of the rules; success rates increased when the goals were only attempted when predicted to succeed.

The authors present the need for properly classified examples from which to train the inductive learning agent; the lists of actions and predicates, as well as plans and desired goals, all must be created in advance by a domain expert. It would be interesting future work to compare the results of the authors' experiments with the results from the scene recognition algorithm presented in this thesis; unfortunately, this paper was discovered after the experimental framework (Chapter 8) was already completed.

Layered Learning

While there is a clear distinction between learning skills and learning rules (and the methods and algorithms used for each), they are not mutually exclusive — in fact, the vast majority of RoboCup clients use what is known as a "layered learning" approach, combining both low-level skill training with a hierarchical application of different types of machine learning.

Peter Stone defines layered learning by four key principles [71]:

1. A mapping directly from inputs to outputs is not tractably learnable.
2. A bottom-up, hierarchical task decomposition is given.
3. Machine learning exploits data to train and/or adapt. Learning occurs separately at each level.
4. The output of learning on one layer feeds into the next layer.

In a typical layered learning approach, basic skills such as ball interception, dribbling, and shooting are learned in one layer, higher-level abilities (such as goaltending,

pass selection between teammates) are learned in another layer, and even higher-level abilities (inter-player communications, formations) at another layer. Each layer abstracts the layers below it, so that a multi-agent strategy layer can perform reasoning based on previously-learned fundamental skills such as passing and blocking.

Each layer can employ a different learning strategy (e.g. hard-coded, scripted, supervised training, real-time calculation). The decomposition of tasks into smaller subtasks allows for more complex behaviour, using learning techniques that can be customized to specific problems.

The underlying architecture of most existing RoboCup clients (particularly the championship-winning ones) consists of several layers of skill ability. A player learns a set of basic abilities, then a higher-level strategy this may consist of formation behaviours, communication with other teammates, player types (i.e. offense, defense), and so on. Many teams also implement a separate RoboCup "coach" agent to oversee the team and give extra information to the players [10].

Teams demonstrating the use of a layered learning model include CMUnited [69], one of the pioneers, TsingHualos [83], which uses a Q-learning algorithm for ball handling skills and an adversarial planning algorithm for strategy, ShaoLing [87] (three layers for strategy, tactics, and individual execution), Dirty Dozen [58], UvA Trilearn [37] and many others. RoboLog [68] uses first-order logic predicates to perform basic spatial reasoning on the raw server data before using a belief-desire-intention architecture to act on this information. Takahashi [74] describes a robot in the RoboCup "small size" league which relies on a multi-layered approach from sensor acquisition to reinforcement learning and high-level strategy selection.

The layered learning model is clearly effective and is the paradigm of choice for

the majority of RoboCup researchers. Its only major drawback, from an automated learning perspective, is the high degree of human supervision required to ensure accurate learning or to directly influence the resultant behaviour of the agent in any of its layers.

3.1.3 Player-Influenced Learning

Another interesting approach, and one that has potential to reduce the reliance on human domain experts, is to consider the behaviour of other existing RoboCup clients. Much as humans intuitively learn by observing other humans, clients that use player-influenced learning strategies are influenced by the actions of another agent. UvA Trilearn [37] uses "Mutual Modeling of Teammate Behaviour" to help an agent estimate the state of its visible teammates, for example to determine whether a teammate is ready to send or receive a pass.

In addition to learning from other agents, there is potential in learning from the behaviour of humans (or human-guided avatars). Several projects consider the use of human interfaces to allow people to control RoboCup agents, including two pure human interfaces by Marlow [51] and Spoelder [66] which allow soccer players to be directly manipulated by a person using a GUI interface. Currently these approaches are limited, since human players respond to the game much more slowly than the software players do. This is partly attributed to the user interface – it takes time to input parameters such as a desired speeds, directions, and powers. ITAS, or "In The Agent's Shoes" [51] is a project designed to increase the response time of a human player by using a layered-learning architecture; underlying skills such as passing are learned (using decision trees) from previously mined data, and the ITAS

interface uses this information to inform the human user of potential pass situations. When the player decides to pass, the SmartITAS agent automatically selects the appropriate kick power and direction based on previous training instances (using a k-nearest-neighbor search).

While these agents use player modeling techniques to dynamically change their own behaviour, the training must still be done offline by a domain expert.

Behaviour modeling of opponents and teammates rely on the assumption that it is possible to use a small set of models as a basis for describing a larger set of behaviours and decisions. Research results (e.g. [67]) indicate that this assumption is valid in many situations. (Note that the choice of the model directly influences the subsequent choice of the machine learning algorithm — and vice versa.) The general problem of agent modeling is discussed in further detail in Section 3.3.

3.1.4 Lessons Learned

From this survey of existing RoboCup clients, we apply the following to our own agent development:

- "Learning by observation" is generally performed in a context of real-time scene recognition (mapping the currently unfolding events to previously stored descriptions), i.e. [67, 37].

- The majority of player-based learning techniques require a high degree of manual fine-tuning of such stored descriptions (i.e. [67, 71]); automated description generation is considered a continuation of the research that has already been done. As most RoboCup researchers are interested in developing high-performance teams for competition, such approaches have not generally been considered.

- The commonly accepted and well-proven agent architecture is a layered-learning approach built on supervised reinforcement learning. Our agent development should follow the same process of training a low-level skills-based layer, which can be followed later by the addition of higher-level learning layers. For simplicity, the initial development goal will be to produce a single-layered learning system combining both skills and strategy. Evolutionary improvements may include additional learning layers to separate skills and strategy (ideally with a common knowledge representation format used between layers, to facilitate unsupervised translation).

- Most existing teams have complex architectures as a result of their layered-learning approaches. While a simple client such as Krislet is not practical for competition, it is ideal as a starting point for new development and experimentation.

- Knowledge representation formats vary, but in most cases are stored externally from the agent's program code.

We have observed that a large part of the complexity of a typical RoboCup client lies in how it models its surroundings and how it represents "knowledge" — of its domain, its surroundings, and its representation of rules and strategies. We now consider several types of knowledge representation formats used both in RoboCup agents and in other domains.

3.2 Knowledge Representation Formats

As the architectures of RoboCup clients become more complex, so do the knowledge representations used within them. In fact, it is generally the knowledge representation format which defines the flexibility of the agent and its learning ability, since the format defines the choice of learning processes which can be used to train the agent.

A distinction should be made between a *data representation* and a *knowledge representation*. The difference is that a knowledge representation should convey not only the raw data but also *infer some semantic meaning*, derived from the data. Generally this new knowledge can be used in (and/or is a result of) a high-level reasoning process. In layered-learning approaches, there may be several representations in use (e.g. a decision tree and a semantic language processor) though some representations may be suitable for use in multiple layers.

Since RoboCup is a simulation of a real-world spatial problem (the game of soccer), we are interested in representations that capture spatial knowledge, given the provided information about the position and velocity of players and other objects on the field. Numerous spatial representations are used to capture a great deal of information including definitions of objects (ontologies), topologies, sizes, distances, relative orientations, or shapes [13]. The RoboCup application domain constrains the selection to representations that support (only) the following key features:

- a specific closed collection of objects (players, flags, ball, etc.) and include two-dimensional position and orientation
- size and shape of objects are generally not important
- semantically rich

- provide structure to the data
- easily generate automatically from RoboCup data
- easy to process or interpret automatically with little or no human interaction

This section provides an overview of some common or prominent knowledge representation formats used both within current RoboCup applications and in other relevant research applications.

Typically, spatial knowledge representations fall into one of the following categories:

- Graphical representations
- Sets of logical predicates
- High-level language descriptors

Murakami [57] cites an objective to (a) develop a description "language" which is used to describe events, objects and relationships between them, and (b) establish a description process by which the raw observed data can be described using the description language. Though Murakami seeks to model human problems (traffic flow and social interactions) the same approach can be taken with RoboCup.

3.2.1 Graphical and Graph-based Representations

Many applications of spatial knowledge representation occur when capturing data from sources such as frames of video. This is common in applications such as face or object recognition, speech recognition, map generation, maze solving or navigation,

etc. [13] Typically the raw data consists of an array or collection of values (perhaps pixel values corresponding to bitmapped images). This data may be processed to extract relevant features, suppress non-relevant ones, search for patterns (such as duplication of objects), or detect edges or lines. After processing the raw data, representations may be in the form of a graph (nodes and directed edges), vectors, diagrams, etc.

Some examples of the use of graph-based representations include the following:

- representations of paths through physical space, such as in a maze-solving algorithm using a Voronoi diagram to plot the best path, or through the use of cellular automata (representing space as a grid of cells) [39].

- various applications relating to image processing in the RoboCup small-robot league including a system for visual homing [75] in which the robot develops homing vectors by examining paths and positions of objects from memory and from an on-board camera; extraction of significant features (edges, landmarks) from bitmaps for various applications [75, 5, 78]. Yamada [81] considers the generation of symbolic data (location of players and a ball) based on the processing of bitmapped images provided by a video camera.

- as a symbolic representation of physical objects, e.g. knots in a rope [33]. Ropes are represented as lines, and their interconnections as nodes in a graph; the symbolic form can then be manipulated to solve spatial problems involving the physical objects.

- as a representation of of sequences of actions, e.g. RoboCup agent actions represented as action graphs [3], which show actions connecting time point

vertices; these graphs can be simplified and analyzed by humans or by a machine learning process.

Graphical representations see most frequent use in the physical RoboCup leagues, which typically use an overhead camera to provide the robots with spatial data. Since the RoboCup soccer server already provides the data in a numeric, parsed form, the "level of knowledge" available directly from the RoboCup server is generally already on par with what is available from these representations.

3.2.2 Predicate Logic Statements

A very common type of spatial representation is the use of logical predicates to describe attributes of objects or relationships between objects. Typically, predicates can describe the following types of information:

- object position (near, far, left, right)
- relationship with other objects (to-the-left-of, behind, in-front-of)
- object state (can-pass, can-kick)

Using predicate logic statements allows for a fairly straightforward input into many decision-making or learning algorithms. Predicates can be used directly in decision tree logic, inductive logic programming or as state descriptors for state machines and planning engines (e.g. STRIPS, BDI).

Numerous examples of predicate logic representations exist, both within the RoboCup domain and elsewhere. While the domains are different, the purpose is largely the same: to generate a description of an event, scenario or entity, given a larger base of data (uncategorized, raw, lower-level, or otherwise). These include the following:

- RoboLog [68] is both a RoboCup team and a library package for client development. It "provides estimates the clients position, provides a history over several simulation steps, keeps track of clients stamina and supports simple qualitative reasoning". The goal of the RoboLog library is to provide low-level basic skills and perceptions to higher-level layers (written in Prolog). The RoboLog library provides translation of raw data into spatial predicates such as *is_between* or *is close by.*

- agents developed based on inductive logic programming [34, 52] (Section 3.1.2 describes a case study of an ILP learning agent).

- interpretation and descriptions of topical maps in GIS applications [42], where maps are described using predicates such as *contain*, *type_of* or *line_to_line.* Maps are translated into their logical meanings, e.g. regions of land types that contain points of interest and are connected by paths.

Knowledge can easily be generated and represented, and knowledge "units" can be composed simply by aggregating the required logical predicates (using typical Boolean logic such as AND, OR, etc.) The drawback, from an automated-process perspective, is that predicates must be explicitly defined and tested for programmatically. Lists of individual predicates can become unwieldy and unintuitive, which can make it difficult to build or verify models. This process is typically done by hand, or if machine-generated, designed for human verification [34].

3.2.3 High Level Language Constructs

The evolutionary next step from working with lists of predicate logic statements is to organize them in a logical way, e.g. by grouping sets of predicates into descriptions of events or situations. A formal language representation is one way to express this by adding structure and support for functions, conditional statements, hierarchies of predicates (rules composed of other rules), plans, etc. Once expressed in a logical language format, lexical parsers can be implemented to "execute" rules; the rules themselves are stored in a human-readable form that is easy to understand and edit (once the language syntax is learned, of course).

Other constructs may include well-defined ontologies [13] or other domain specific devices (libraries of graphical shapes, rules, bounds [14]). Language constructs include "Q" (based on Scheme), and its related form of "Interactive Pattern Cards" [57], used for describing how agents should behave when dealing with humans or other agents. Murakami describes defining the vocabulary, establishing cues, actions, and scenarios, extracting patterns, followed by evaluation. The result is a set of written scenarios ([57] cites a "follow-me" method for evacuating a building). Similarly, Yue [86] describes the use of scenarios as behaviour patterns and describes the use of a specification language called GIST to model a public library and the scenarios that may occur (patrons borrow and return books, resource usage, etc.) An algorithm is then used to generate scenarios from simulation traces. Yue notes specifically that "the program only works on small examples due to oversimplified search strategy".

RoboCup itself makes use of a language called CLang [10, 58] through which coaches may communicate with their teams, sending them information (from which agents make their own decisions) or advice (specific instructions that agents should

follow). CLang does not contain much of a library of logical predicates, so an enhancement, SFL, was created [58].

Case Study: SFL and CLang

SFL, or the Strategy Formalization Language [58, 67] is a "language representation of features", and an extension of CLang, the standard RoboCup coach language used by coach agents to communicate with their teammates [10]. CLang supports description of rules or instructions, such as "if our player 7 has the ball, he should pass to player 8 or player 9" [38]. SFL extends CLang with further primitives such as *closestPlayerToBall* [67].

Steffens describes the use of SFL to model the high-level behaviour of soccer agents [67] and notes the following characteristics of SFL:

- it is well suited to providing high-level descriptions of opponent models (opponent agent behaviour)
- it describes attributes that are externally observable (ignoring any internal decision making processes)
- it cannot model all details at low levels (i.e. there is no way to specify dribbling, or kicking, individually)
- it is able to make distinctions between entire teams on a tactical level
- it supports a varying degree of granularity — it can model very general rules as well as rules describing very specific situations
- it does not support the description of iteration or recursion

Figure 3.1 shows a rule written in SFL (in the form of advice given to a team) which states that if the ball is under the control of any opponent, the closest player to the ball should intercept it with maximal speed.

```
(advice
   (6000
      (and
         (bowner opp {0})
         (playm play_on)
      )
      (do our { (closestPlayerToBall our)}
            (interceptball 100))
))
```

Figure 3.1: A rule in SFL instructing players to intercept opponents controlling the ball, if they are the closest.

SFL is clear, concise and logical — ideal attributes for a description format used in a situation-matching algorithm. However, the complexity of the language suggests that human experts are required to generate rules of any complex nature (this is how team profiles are generated in [67]; the recognition engine only parses it, and automatic generation is cited as difficult future work). There is no straightforward conversion process from lower-level data directly into a syntactically-correct SFL representation. SFL also does not easily lend itself to learning approaches. With these limitations, SFL is best suited as an output format data structure rather than an input or intermediary format.

3.2.4 Lessons Learned

There are numerous forms of knowledge representations which range in abstraction level, complexity and capability. Most are highly specific to their domain, typically necessitated by their expressiveness (the more expressive, the less general the representation must be). Some formats are designed to be generated by automated processes, and others only to be interpreted. Most of the more complex, feature-rich representations examined above rely on a domain expert to generate and/or verify.

In selecting a knowledge representation for a RoboCup imitative agent it is important to note that the RoboCup soccer server already encodes much information within the messages sent to the players. Messages are essentially already symbolic and can be processed into forms such as generated predicates or graphical representations. Since much of the work is already done (typical agents must already process messages from the server anyway) it may be beneficial to use a knowledge representation format that expands (and is based on) on the data structure already provided by the RoboCup server.

One key application of using learning techniques and semantic knowledge representations is the ability to create models of other agents. The next section describes existing efforts in this area of research.

3.3 Agent Observation and Modeling

The ability for an agent to observe another and learn from these observations relies on the ability to create a model of the observed agent's behaviour. There are two primary purposes for agent modeling:

- Observing agents and matching them to previously-generated profiles (as in modeling and reacting to the behaviour of adversaries or opponents)
- Dynamically generating models of agent behaviours (as in classifying behaviour of a complex system into a simpler representation).

This level of behaviour modeling is high-level and beyind the scope of this thesis; however, the following sections provide a brief overview of these techniques since they show promise as direction for future work development.

3.3.1 Agent Behaviour Observation

Agent modeling has much practical use in the RoboCup environment. Typically, specific behaviour models can be used to profile adversaries (such as RoboCup opponents) or to assist an agent in predicting its teammates' behaviour and assisting it to accomplish its goal.

Newer versions of the CMUnited agent, for example, incorporate a technique called "ideal model-based behavior outcome prediction" to model teammates and opponents and determine whether to shoot, pass, or dribble based on its prediction of the other player's intent [72]. Timo Steffens [67] describes a high-level strategy selection approach in which a team strategy is dynamically chosen based on "opponent behaviour modeling" — the team maintains a library of strategies, descriptions of opponents it has successfully beaten, and which strategy defeated which opponent. As the team plays, it models the behaviour of its opponent; if it is recognized, the team begins executing the strategy known to have previously defeated that opponent. Opponent profiles are generated (and represented in SFL) which uniquely characterize particular opponents and describe a strategy for counter-attack. The agent implements

a statistical recognizer which detects when a profile is matched with a high enough confidence, and reacts accordingly. Han and Veloso [30] implement a similar approach with Hidden Markov Models, creating models for actions such as Go-To-Ball, Intercept-Ball or Go-Behind-Ball. The recognizer algorithm dynamically generates probabilities of matching each of the predefined action models.

In each of these cases, descriptions of agent behaviour (e.g. SFL and Hidden Markov Models) are complex and must be hand-generated; Steffens describes the automatic generation of player models as future work (some of which Matsui [52] attempts to address using ILP).

3.3.2 Generating Models of Agent Behaviour

Dynamically generating a model of an agent may be useful when dealing with unknown adversaries (where the human domain expert is unavailable or unwilling to assist), or when modeling behaviour which may not yet be explicitly defined (such as animal and insect behaviour).

Typically, the observed agent is assumed to act according to a known set of behaviours which is reactively chosen according to the agent's state [30], and the observer must recognize the behaviour and determine a model of the agent's internal state. Common approaches include the use of Hidden Markov Models (used in modeling of RoboCup soccer agent behaviour [30] and in modeling the behaviour of honey bee dances [25]), or other state-based models such as deterministic finite automaton (e.g. to model opponents in two-player games [61]) or finite state automata [45].

Feldman uses a combination of Hidden Markov Models and the k-nearest-neighbor classifier. Raw motion data (coordinate positions and movement vectors) from honey

bee movements are identified (arcing left, arcing right, straight, waggle, etc.) using the k-nearest-neighbor classifier (using a previously-labeled training set); the output of the classifier is fed into the HMM algorithm. Feldman warns that while kNN classification works well, it has a limitation in that it "considers each data point individually, without considering the sequence of data points as a whole" [25]. The author describes a "sensitivity analysis" stage in which all attributes were tested and only the features that resulted in the highest accuracy were kept, essentially weighting the attribute data.

3.3.3 Lessons Learned

We have examined several approaches for modeling and recognition of agents, involving high-level descriptors of agent behaviours. Typically the behaviours are recognized from previously-generated models (teammate or opponent behaviour modeling), or state-based models are created based on observing and classifying actions and tasks executed by the observed agent. Applications vary greatly from modeling agents in the RoboCup domain to other game and agent domains to modeling of natural behaviours; each application is highly domain-specific. In each case, the modeling process requires semantic feature definition or extraction (definitions of a known set of distinct behaviours); models themselves are either written by hand or generated only after significant amounts of data processing.

Behaviour modeling approaches are, by definition, most well suited for work with higher levels of behaviour representation (they rely on a pre-existing basis of behaviour definition, such as actions or states). These would be better suited for future development after a basic level of imitative behaviour is already established (e.g.

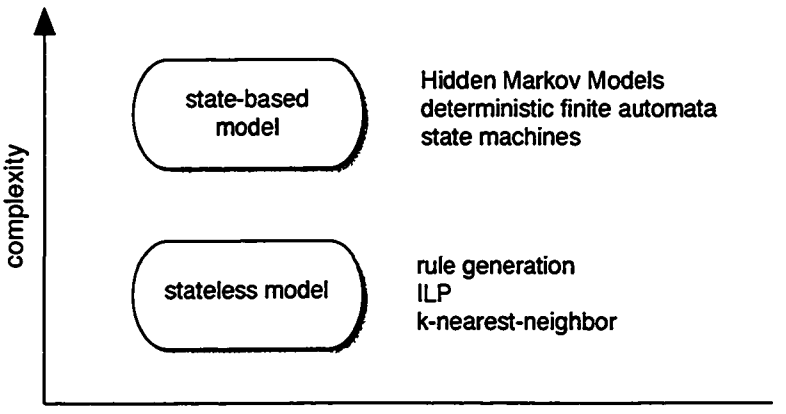


Figure 3.2: State-based and stateless behaviour modeling approaches.

higher "layers" of learning). Figure 3.2 depicts this structure, which also serves as a roadmap for the imitative agent development. Our first goal is to imitate the behaviour of a stateless agent; once this has been established, modeling more complex agent behaviours may follow.

The first steps of Feldman's honey bee modeling methodology [25] are well-suited for adaptation to the RoboCup domain, as they involve a capture of raw visual data, interpretation into spatial predicates, and matching with previously-known actions through k-nearest-neighbor search. These actions have a strong parallel in the RoboCup domain, and Feldman's work even suggests some direction for future work using Hidden Markov Models to build on top of the kNN classifier.

3.4 Situation Recognition and Matching

The data structures and algorithms described to this point have reached a high enough level of abstraction, so we may now consider high-level descriptions of agent

behaviours (typically described as having goals to achieve, and steps taken which decompose into the "actions" an agent takes). It has been well established that behaviours can be described and stored (e.g. [86, 68, 67]), and subsequently be detected ([30, 67, 72]). We consider some techniques used to "recognize" agent behaviours (matching a behaviour to a previously-known one).

Recognition tasks can be categorized into:

- token matching
- spatial scene recognition
- temporal event recognition

3.4.1 Token Matching

One common matching application is that of tokens such as characters within strings. This has use in applications such as natural-language processing, spell-checking and other text processing, speech signals, and interpretation of DNA [6].

Typically, string comparisons are made in which a key string is matched with another from a set of search strings, or some measure of "distance" is formed between two arbitrary (assumed similar) strings. Buss describes an application of a bipartite weighted-graph matching approach to this problem [6], while other matching approaches include substitutions, insertions, transpositions, deletions, or other syntactic pattern recognition approaches [59].

A token-matching approach may be applied to RoboCup if a description of the visual data may be encoded in a representation that can be expressed as a string of characters (stream of bytes).

3.4.2 Scene Recognition

Lemaire [43, 44] defines the problem of scene recognition as follows: given a description of the perceived scene, and a description of the expected scene, recognition of the scene is the matching of both descriptions. Lemaire (writing in the context of describing situations to a robotic arm with a camera and a manipulator) describes a scene as written in a high-level natural language without numerical input, in fact using spatial predicates such as "in_front_of" or "connected_on_the_left_to".

Matching of scenes depends on establishing a set of confidence values that particular descriptions within a given scene are true (e.g. how close the perceived scene is with the description of each expected scene). These may be calculated in a fuzzy logic system [43, 44], using neural networks (Shibata [65] describes a robotic vision system that detects scenes in images), or through comparison of invariant features in each scene [11].

In the RoboCup domain, a "scene" may describe the spatial orientation of objects on the soccer field; a recognition system would seek to match scenes from a library with the currently-perceived scene from a soccer playing agent. Scene descriptors include positions of objects in space, which suggests that geometric matching methods may be applicable. The k-nearest-neighbor classifier may be used to find scene matches: Baltes [2] describes an empirical evaluation comparing different algorithms in a simple pursuit-and-evasion game, and concludes that a scene-based k-nearest-neighbor learner performs best overall (compared to random, optimal, genetic algorithms and a reinforcement learner).

Such an approach relies heavily on a valid distance metric; Schroeder [62] states that "the choice of distance is paramount for analysis". Schroeder describes the

use of Euclidean and Minkowski distance applied to the capture and processing of multivariate data for multi-agent visualization.

In addition to describing spatial configurations, scenes have application in the design, development and documentation of software agents (or even non-autonomous, complex software systems; for example, use-case scenarios in object-oriented programming [20]). The behaviour of an agent can be described in terms of how it reacts in given scenarios; this is just as applicable when designing a new agent as it is when trying to re-engineer or reverse-engineer the operation of an existing one [86, 20]. These applications may also require temporal events to be defined as part of the scenario description.

3.4.3 Temporal Event Matching

A complement to the detection of scenes (matching of spatial attributes) is the detection of sequences of events in time. Applications include the behaviour specification of software systems (as in [20] above), detection of error conditions from alarm-driven events typical in supervision of networks [4, 18, 17] (a sequence of temporal events is referred to as a "chronicle"), interface agents [22, 54] (which detect patterns in events caused by the user). The general problem of pattern recognition in time series data has application in any domain in which sensors produce information on critical systems, such as medical science, manufacturing, and aerospace [55, 7].

Event matching algorithms generally fall under two categories: pattern-search-tree recognizers which consider exact matches or matches with a degree of "distance" and the similar "feature decision tree" structure (Kaminka [35], Caraça-Valente [7], Morrill [55]), or state-based recognizers such as the chronicle-recognition engine [4,

18] (in which, when events arrive and are processed, possible chronicle matches are maintained until a contradictory event arrives and kills them) and Hidden Markov Models (as part of the RoboCup agent behaviour detection algorithm in [30]).

Like spatial matching, temporal event matching relies on the pre-existence of pattern libraries or definitions (though some systems provide assistance in discovering these from offline data processing [4, 55, 7] and others provide facilities, such as interface agents [22], through which these are automatically created).

3.4.4 Lessons Learned

A simple behaviour recognition can be established by matching descriptions of objects at a given time (scenes, by Lemaire's definition [43, 44]) using algorithms such as k-nearest-neighbor [2, 25]. The key to successful recognition is a suitable distance metric [62].

Schroeder [62] and Feldman [25] describe a data generation process which may also be applied to the RoboCup domain, in which raw data is observed, captured, and processed. Such a process could generate the library of scenes from which recognition takes place.

Combining this scene-generation with the behaviour recognition would result in a decision process for an agent based on the knowledge captured in the scenes (similar to the simple agent behaviour-programming demonstrated by Baltes [2], who used pre-generated scenes coupled with the kNN algorithm). If the scene generation from observed behaviour can occur in an automated process, this satisfies the goal of creating a hands-off training process for a RoboCup agent.

There is strong potential for temporal event matching techniques in an imitative

recognition client, since the RoboCup domain is both spatial and temporal. Incorporating the detection of not just single scenes but sequences of scenes is a logical next step for the learning agent to consider. Initial steps should consider scenes individually before considering groups of scenes.

3.5 Imitative Learning

If high-level behavioural knowledge can be extracted from the observation of other agents, it should be possible to feed this knowledge to an *imitating* agent. The result is an agent that learns to "emulate" the behaviour of the observed agent. This happens to some extent when an agent uses machine learning techniques to derive rules for low-level skills (passing, kicking), using observations of "correct" situations as training data.

Some efforts have been made to develop agents that learn directly from the influence of other existing agents. The "Zed" agent [79] and "Agent007" [28] are two attempts by students at Carleton University to learn an entire agent strategy (both using Krislet as the observed model) using only commonly available ("off-the-shelf") machine learning tools. The process in each case is the same: capture logs describing Krislet's behaviour, convert them into a representation of knowledge, and apply machine learning approaches to generate rules for the new agent to follow.

The "Zed' client uses decision tree learning to model and imitate Krislet's behaviour, and "Agent007" does the same using neural networks (the authors also evaluated the use of Naive Bayes and Support Vector Machines, but found neural networks to provide the best performance). Xie et al demonstrate that in some circumstances, conventional learning and modeling methods are sufficient to describe

the high-level behaviour of a RoboCup agent, but with limitations in performance and complexity (in fact, the same limitations as imposed on simple clients, Section 3.1.1). In the case of the Zed client, much intervention was required to prune the learned tree and provide positive training instances to help the learning process. In the end, however, the result was an agent that bore a strong resemblance to the Krislet agent. Similar results were reported from the "Agent007" group, though performance was not as strong as the original [28]. In each case, the agents appear to play in a style similar to Krislet (qualitative performance) though the agents tended to be slightly slower, do not always play correctly, and when pitted against an actual Krislet opponent, scored fewer goals.

Both projects capture log data, then define easily-recognizable attributes and use a utility to convert the data into the ARFF[2] format. The Weka [27] library of machine learning tools (decision trees, neural networks, support-vector machines) are used to determine rules correlating the input data with the output actions selected by the Krislet agent. A new RoboCup client is then created which implements the learned rules.

Both projects successfully developed decision rules that correspond to those programmed into Krislet. However, the method appears to break down quickly when the behaviour becomes more complex; the "Guru" client [47], when attempting to perform the same methodology on more sophisticated clients, failed to gain any useful decision rules from the machine learning process. The authors suggest that performance would improve if the same attributes are used in the decision-learning process

[2]An ARFF (Attribute-Relation File Format) file is an ASCII text file that describes sets of data sharing common attributes. ARFF files were developed by the Machine Learning Project at the Department of Computer Science of The University of Waikato for use with the Weka [27] machine learning software.

as that of the original agent. Given the sheer number of possible attribute combinations, this could only be possible given a detailed decomposition of the original — and this reduces to the previous problem of supervised learning of low-level skills.

3.5.1 Lessons Learned

The attempts of [47, 28, 79] are notably limited, particularly given the multi-layered and sophisticated behaviours of most RoboCup agents. However, given the variety of methods for data capture, behaviour representation, scene recognition, object matching, etc. it may be possible to produce better results given a combination of these higher-level techniques.

3.6 Conclusions

This chapter has examined a range of methods for capturing, representing, finding and working with knowledge representations in the domain of the RoboCup soccer simulation. While individual strategies and methods vary, the key to every RoboCup agent is its ability to store knowledge in a spatial representation format and use it to act in real-time. In some situations, agents are capable of developing models and observing the behaviour of other agents. There have been several attempts at combining the two approaches — developing a knowledge representation by observing other agents — with the intent of emulating observed behaviour from other agents.

A common limitation is present in many of the learning and training approaches described in this chapter: the strong reliance on a human domain expert. Most current RoboCup agent approaches require a great deal of human intervention, either

to hard-code an algorithmic approach (such as an opponent model) or to create a set of ideal training scenarios in order to train a lower level skill and validate the results.

Existing efforts to learn agent behaviour directly from machine learning [47, 28, 79] or ILP [52] approaches are moderately successful. Models of simple, stateless agents are possible, though still with a need to preprocess data prior to training the agent.

Stone [70] and Steffens [67] describe higher-level behaviour model representations and both express that the automatic generation of agent behaviour models is an interesting "next step". Potential solutions come in the form of the many tools available to describe and recognize scenes and events. By using this "toolbox" of different approaches, it is our goal to attempt to develop a system where RoboCup agent behaviours can be observed from other agents, learned, and then applied directly into a new imitative RoboCup agent, using the tools described above.

Our development will use the following approaches:

- A simple hard-coded agent (Krislet) as a development template for creating a new RoboCup software agent
- Stateless behaviour modeling
- Model generation from capture, processing of raw data from observation of RoboCup agents [62, 25]
- Matching of descriptions of objects at a given time (scenes, [43, 44])
- k-nearest-neighbor search [2, 25]
- Selection of a suitable "distance" metric for scene matching

By combining aspects of these different approaches, we hope to gain flexibility and overcome the limitations of individual approaches.

The following chapters refine these ideas further; Chapter 4 describes the formal methodology for the capture, data modeling and recognition process. Chapter 5 describes the development a scene representation format, and Chapter 6 describes the algorithm used in the scene matching process.

Chapter 4

Methodology

> "Like all Holmes' reasoning," Dr. Watson says, "the thing seemed simplicity itself when it was once explained."
>
> *Sir Arthur Conan Doyle*

The previous chapter motivated the use of an imitative learning system for automatic training of a RoboCup agent, and provided an overview of the relevant areas of research necessary to develop such a system — spatial knowledge representations, RoboCup client architectures, learning algorithms, etc. Sections concluded with notes about which methods showed promise and made suggestions about what contributions could be made toward our stated purpose.

This chapter now describes the overall methodology — the process by which data is captured, represented as knowledge, and ultimately used by the scene recognition agent. Details about the actual implementation, as well as discussions of the scene recognition problem and the chosen representation format, are left for Chapters 5, 6 and 7.

4.1 Methodology

Our objective is to develop a soccer-playing RoboCup agent with the ability to draw from observations of other RoboCup agents and use this knowledge to guide its own decision-making process, all with limited or no domain expert (i.e. human) intervention. The agent should be able to derive its own knowledge base and training all from its observations of other agents.

The completed framework consists of two major components: the offline learning component, and the RoboCup agent itself. The complete process includes the following objectives:

1. *Perform data capture from logs generated by existing RoboCup clients.* Intercepted communications between existing players and the server are captured to a log file. The resulting logs capture the game as seen from *each player's* point of view.

2. *Store the captured data in a spatial knowledge representation format.* We store the captured data in a format that represents individual "scenes" — snapshots of an agents behaviour at discrete times.

3. *Apply learning algorithms to the stored data (where possible).* We may attempt to interpret the behaviour of the recorded agent, using higher-level logical constructs to increase the level of generalization.

4. *Apply a search algorithm in the agent to compare real-time situations with previously-stored ones.* Using the set of stored scenes as its knowledge base, the new RoboCup client should respond to each situation by using the same actions that other RoboCup clients used in "similar" situations.

Figure 4.1 shows a block diagram of these tasks. Of the components shown in the figure, some are reused from existing software (see Chapter 7 for implementation details). The shaded components represent contributions from this thesis.

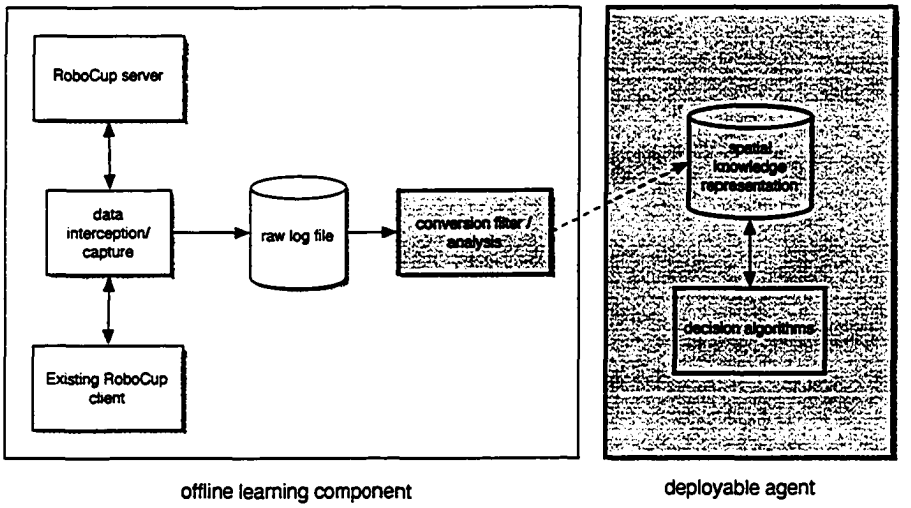


Figure 4.1: The general methodology for training and developing RoboCup agents. (Shaded components represent contributions from this thesis.)

Once the process established, it may be used to train the RoboCup player agent using a variety of input data sources (existing RoboCup clients). The resulting agent performance should be directly based on the observed inputs (the resulting trained agent should, in fact, exhibit behaviour similarities with the training agents) with little or no changes or adjustments to the process itself. This can be measured both qualitatively (observed similarity) or by using quantitative statistical measures.

4.2 Assumptions Made

An ideal candidate to learn from is one which performs well in RoboCup simulation games or otherwise clearly demonstrates domain "knowledge". By definition most publicly available RoboCup teams meet this critera, though clearly preference may be given to "accomplished" teams such as champions from RoboCup World Cup games.

For this method to work, it is assumed that the actions of the observed player can be derived directly from the visual inputs it sees at the same time cycle. This assumption places limitations on the immediate success of the imitative algorithms, since it cannot account for agents which have other underlying motivations not directly or completely tied to the status of their environment:

- Agents that make significant decisions based on internally-stored prior memory of previous events (though future work will seek to address this),
- Teams that rely heavily on inter-player communication — since messages are encoded in a proprietary format, it would be difficult to decode the semantic meaning of messages and how the player reacts to them [51],
- Teams that employ coaches — similarly, instructions from the coach would affect the player in ways that would not be observable nor relevant, since the coach will not exist in the newly trained agent.

The impact of this assumption also varies with the complexity of the agent being observed. Ideal candidates for observation are agents that are stateless and have simple goals which are directly tied to the positions of the objects in its environment.

Chapter 5

RoboCup Scene Representation

To model an object is to possess it.

Pablo Picasso

This chapter describes the chosen scene representation format, which will be used to represent spatial scene information from RoboCup games. Chapter 6 will later describe the algorithms used to operate on scenes, while Chapter 7 describes the actual implementation of the scene representation format and algorithms.

5.1 Agent Behaviour Modeling

The initial development of the imitative behaviour framework assumes a direct correlation between the actions of an agent at a given time t and the inputs received at that time. Recall from previous chapters that the observed agent is treated as a black box implementation of a function f relating inputs $x_1, x_2, \cdots, x_n$ (equation 1.1):

$$y = f(x_1, x_2, \cdots, x_n)$$

Adding the time constraint gives rise to equation (5.1):

$$y(t) = f(x_1(t), x_2(t), \cdots, x_n(t)) \tag{5.1}$$

(Future work will attempt to consider factors including an agent's internal state or history, actions and inputs in preceding time periods, etc.)

The behaviour function f can be approximated by capturing the values of the inputs and output at various times. We define P as the set of these captured values:

$$P = \{p_1, p_2, \cdots, p_t\} \tag{5.2}$$

Each p_t is a tuple $[y(t), x_1(t), x_2(t), \cdots, x_n(t)]$. The set P becomes a training set for a learning or recognition algorithm which yields an approximation for f.

5.2 A Scene Representation For RoboCup

In the RoboCup soccer simulation domain, each data point p_t describes states of objects $(x_1, x_2, \cdots, x_n)$ at time t, and the agent's response $y(t)$. Each tuple thus describes a discrete-time snapshot of the soccer field, which we define as a *scene.*

Our goal is thus to define a data structure which represents scenes. In this situation, a scene should represent the point of view of a given player, showing the locations of objects (other players, the ball, goal, etc), from that relative perspective. The scene should also store the actions (if any) taken by the soccer-playing agent at the corresponding time.

As it turns out, the nature of the communication format between RoboCup clients

and the soccer server makes it fairly straightforward to capture scenes directly from messages sent by the soccer server.

All visual information is received by the soccer agent in the form of an encoded see message sent from the RoboCup server (Figure 5.1). Each message includes a timestamp and a list of all the objects visible by a client agent. Each object is given a type (the ball, goals, players, flags and lines on the field) as well as attributes including its distance and direction from the client. The accuracy (and in some cases availability) of these attributes depends on their distance from the client and is subject to a random noise model. For example, a player in close proximity to the soccer agent may be described by its team name and its uniform number; if a player is too far away, the uniform number (and possibly even whether the player is a teammate or opponent) may be unknown. Players have limited fields of view, beyond which objects are "seen" only if they fall within a certain proximity to the player, and with limited detail [10].

```
(see 20 ((flag c) 45.6 0) ((flag c t) 69.4 -24) ((flag r t) 107.8 -1)
((flag r b) 92.8 37) ((flag g r b) 92.8 20) ((goal r) 94.6 16) ((flag
g r t) 96.5 12) ((flag p r b) 75.2 28) ((flag p r c) 79 13) ((flag p r
t) 86.5 0) ((flag p l b) 4.3 -11 -0.258 -0.8) ((ball) 44.7 0) ((player
Poland) 36.6 7) ((player) 60.3 -28) ((player Poland) 44.7 0) ((player)
    81.5 3) ((player) 66.7 -6) ((player) 73.7 10) ((player) 90 -1)
            ((player) 73.7 11) ((line r) 106.7 -59))
```

Figure 5.1: A typical see message sent by the RoboCup simulator soccer server.

Each see message thus completely describe the environment surrounding each client agent, as seen from the perspective of that agent, at any given time. A typical RoboCup agent will receive these status and environment messages, at least once per

simulator time cycle, and send a command back to the server in response.

The rules of the Simulation League stipulate only one action command (Table 2.2) per player will be executed during a given simulation cycle[1] [10]. If these actions are also mapped into scenes, the result is a set of ordered scenes which describe the visual input and the corresponding action from the player.

5.2.1 Scene Definition

We define I as the set of visual inputs to the agent at a given time t:

$$I_t = \{x_1, x_2, \cdots, x_n\} \tag{5.3}$$

Let R be in the set of commands an agent could send in response to these inputs (for simplicity we consider only actions that physically affect the game):

$$R \in \{turn, turn\_neck, catch, dash, kick\} \tag{5.4}$$

We define a "scene" as the aggregation of the visual inputs to the agent and the response command at time t:

$$S_t = \{I_t, R_t\} \tag{5.5}$$

An entire game is thus represented as a collection of scenes S:

$$S = \{S_0, S_1, S_2, \cdots, S_n\} \tag{5.6}$$

[1]If more than one action command is sent by the agent, the rest are ignored. Status-request commands may be sent as often as desired.

(where $n = 6000$, the typical number of simulation cycles in a game). The complete set S provides a "play by play" description of one players behaviour during the entire game, showing all of its reactions to the changing environment around it.

A RoboCup scene is thus a direct translation from an individual see message from the soccer server (and its corresponding action-response). Using the information provided from the message, a scene contains a list of all the objects within the viewable area of a RoboCup player client at a given point in time. If plotted graphically, such as on a RoboCup monitor client, the objects surrounding a RoboCup player might resemble the diagram shown in Figure 5.2.

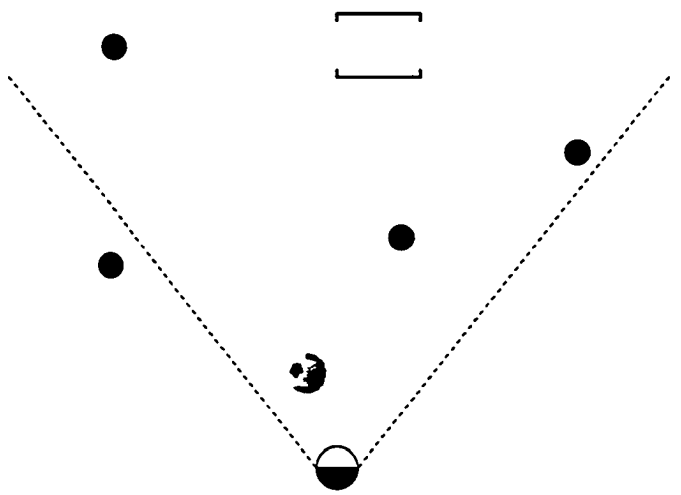


Figure 5.2: Objects around a RoboCup client, as seen from the clients perspective. Dashed lines represent the edges of the agent's view cone.

5.3 Generation and Manipulation of Scenes

The RoboCup environment lends itself well to a scene-based data representation since the data from the simulation server is, conveniently, already described in sets of time-stamped visual data. The initial translation from a captured raw log file to a scene representation should thus be very straightforward.

Aside from ease of implementation, the concept of a spatial scene is intuitive. Human observers of the soccer simulation (typically on a soccer monitor GUI) are already familiar with the visual display of the objects on the field, resembling an actual soccer game in appearance. Many soccer monitors and log-visualizer tools (i.e. [29, 12]) provide facilities for pausing, rewinding, forwarding, stepping through pre-recorded games — actions which are equivalent to displaying individual, consecutive scenes.

Scenes can be as simple as a logical way of interpreting the existing raw captured log data, or they can actually be written as a new, higher-level knowledge representation derived from the original log data. Once such a format is generated, the scenes can potentially be manipulated in a number of ways. The scenes can be written in ARFF format, for example, and machine learning algorithms can be applied directly at this point. However, the use of a logical construct such as a scene also provides a convenient atomic unit through which data can be further processed at a higher level. For example, scenes can theoretically be manipulated by reflection (Figure 5.3) and addition (Figure 5.4). These actions introduce new logical correlations between two scenes — expressing, for example, that an action involving an object in the left side of the agent's view has a corresponding, possibly equivalent, action involving the same object if it was in the right side, in the case of a reflection, or that a set of

scenes might contain common elements (scenes made up of subsets of objects).

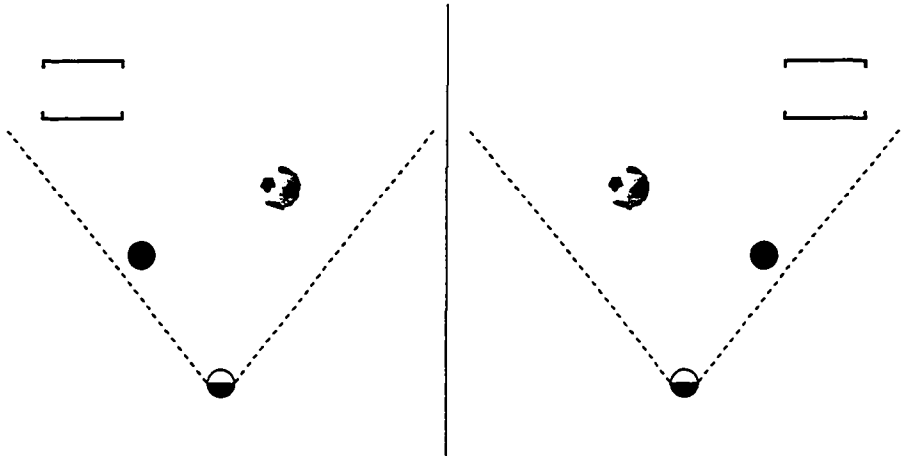


Figure 5.3: A scene and its horizontal reflection (left and right sides transposed).

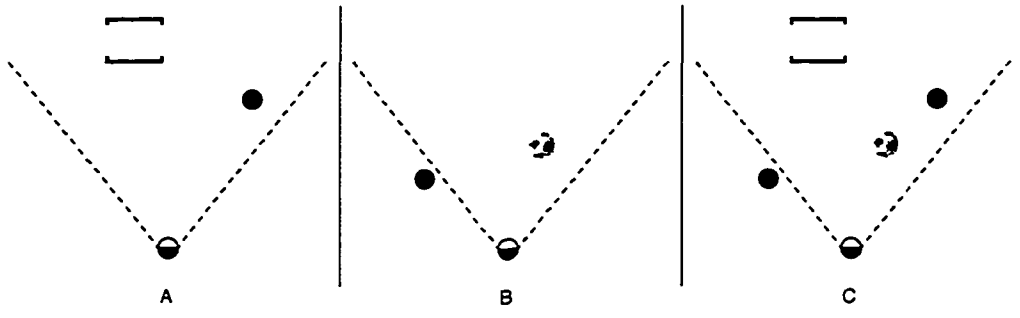


Figure 5.4: Scene addition: Scene C is an aggregation of objects from scenes A and B.

The concept of a scene is implementation-independent, although this thesis will describe one particular implementation using Java objects (see Chapter 7).

5.4 Scene Discretization

A level of generalization can be introduced by partitioning the space surrounding the player into a set of regions. These regions serve to "compartmentalize" objects in the space around the player. Suppose a player has a field of view of v degrees[2]. The field of vision may be divided into n segments, each pie-shaped segment covering v/n degrees of view. An object can now be described as being located in any one of these n "wedges".

Introducing the notion of these wedges has the effect discretizing the data that makes up each scene. Suppose, for example, that the field of vision is divided into five regions, representing the general directions of extreme left, near left, center, near right, and extreme right (all relative to the direction in which the player is facing). An object positioned 90° to the right would then be classified as belonging to the "extreme right" region. This effectively introduces a new set of semantic information that was not previously available, since objects can now be described in these relative terms.

These regions can be partitioned even further by examining the distance from the player to each of the objects. Object distances can be described in general terms; for example, objects could be defined to be "close to", "near", and "far away from" the player. In such a situation, the space surrounding the player can be partitioned into three regions, represented as concentric circles around the player. The radius of each circle defines the boundary distances.

[2]Since the RoboCup soccer simulator server manual [10] defines situations in which objects outside of the viewable area can be noticed by the agent, it may be important to consider the full 360° view instead of just the server-defined viewable area.

Combining these two sets of partitions results in a division of the space surrounding the player into fifteen separate regions. Figure 5.5 shows the effect of this discretization on the same view as previously shown in Figure 5.2. The introduction of these regions allows the classification of objects by their orientation to the player (somewhere between extreme left and extreme right) as well as by their distance (somewhere between close and far). These are, essentially, spatial primitives that can later be used to perform reasoning on a scene.

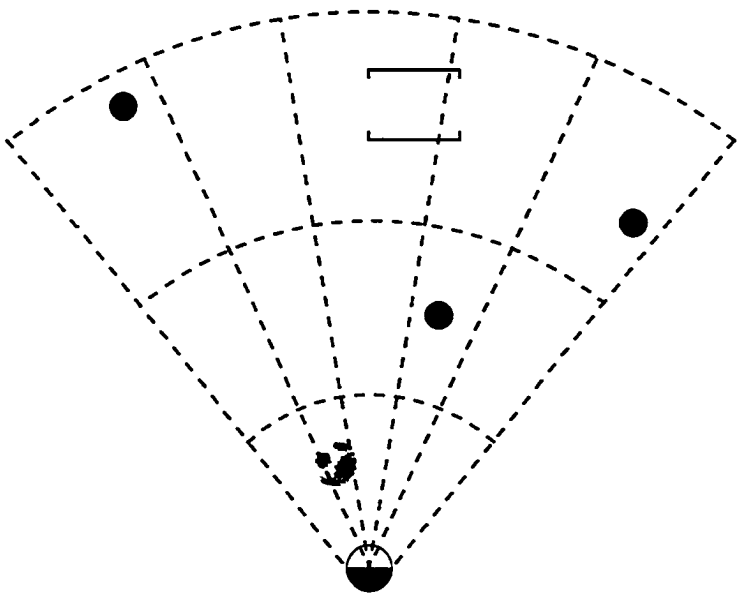


Figure 5.5: Introducing region-discretization of objects around a RoboCup player.

5.4.1 Advantages of Region Discretization

Applying such a discretization of the scene representations can provide the following advantages:

- Object matching can be done more easily since the discrete regions allow the notion of "similar" location (i.e. if two objects fall within the same region). This may increase the probability of obtaining an optimal distance (ideally 0, a scene match), which puts a stop to the scene recognition algorithm sooner.

- Different scenes each with objects in "similar" locations could be classified as being very similar and possibly redundant; this could reduce the number of total scenes needed in the training set. This is particularly important since the recognition algorithm must be optimized for speed (ideally faster than one simulator time cycle).

- The concept of a discretized distance (i.e. adjacent, nearby, far away) is not only intuitive, it is a distinction also made by the RoboCup soccer server (see Section 5.4.3). Scene-matching algorithms must take distance into account since many rules will depend on this information (e.g. a ball cannot be kicked unless it is adjacent).

- Logical predicates can be extracted from examining objects classified into regions; this can later be used in a spatial reasoning algorithm. For example: if b is between a and c, and c is between b and d, then c must be between a and d.

- The scene can be represented graphically in an intuitive human readable form, which also facilitates scene editing or creation (using, for example, a GUI tool). The structured nature of the scene supports data storage using a structured format such as XML.

- By allowing a variable number of discrete "slices", the scene representation supports a configurable trade-off between complexity and accuracy.

- The scene discretization concept borrows heavily from machine vision techniques, drawing particular parallels to the problem of converting raw pixels in a bitmap into discrete values. As such there is potential to apply various types of algorithms based on vision processing such as feature extraction, pattern recognition, suppression of non-essential features (such as, in some situations, lines on the field).

5.4.2 Scene Table Representation

Previous representations of scenes have been depicted as graphical plots of objects in the 2-dimensional space around the player. This is facilitated by the fact that the RoboCup soccer server provides polar coordinates (angle and distance) to each object.

However, once region discretization is applied, a new representation is possible. The scene can also be represented as a table, where the angle regions are represented as columns and the distance regions are represented as rows. Table 5.1 below illustrates this, showing the same scene previously seen in Figure 5.2 and Figure 5.5. (These are rather colloquially referred to here as "visiontables".)

Table 5.1: The scene from Figure 5.2, expressed as a table.

	Extreme Left	Left	Center	Right	Extreme Right
Far	Player		Goal		Player
Near			Player		
Adjacent		Ball			

(The example table simplifies by describing only one object in each table cell; in a real world situation there are likely to be several objects in each region. Each cell

would contain a list of objects contained within it.)

5.4.3 Boundary Values

All previous scene examples have used a discretization featuring five radial slices and three distance regions, resulting in fifteen discrete regions (or fifteen table cells). These values are chosen largely for convenience; scenes could theoretically be defined using any level of discretization with d radial slices and k distance regions. The scene is divided into n total regions, $n = d \cdot k$. Each object can be labelled with a location coordinate in the form (x, y) where $0 \leq x \leq d$, $0 \leq y \leq k$.

Increasing (d, k) has the effect of increasing the specificity of each stored scene, but (by definition) this reduces its generality, increasing the number of discrete regions and making it more difficult to visualize and to find matching scenes during the later scene recognition process. As $d \rightarrow \infty$ and $k \rightarrow \infty$, the discretizated scene becomes equivalent to the original continuous space.

Thus there are two factors which immediately impact the effectiveness of a discretized scene representation:

- The number of regions to generate (values for d and k)
- The boundary values used to discretize continuous-form coordinates to regions

The problem is simplified by applying some knowledge of the domain and its implementation. The RoboCup soccer simulation communications protocol defines its own boundary distances to determine what distance a ball is too far away to be kickable by the player (*kickable_margin*), or at what distance a player is too far away to identify (*unum_too_far_length*). (Figure 5.6) These constants are user-definable in

a server configuration file, though the net effect is generally the same — RoboCup defines a set of distance boundary values which have some logical meaning. These may be very appropriate to use as boundary values when creating scenes.

Boundary values could also be generated experimentally or from the output of another learning process, e.g. [51].

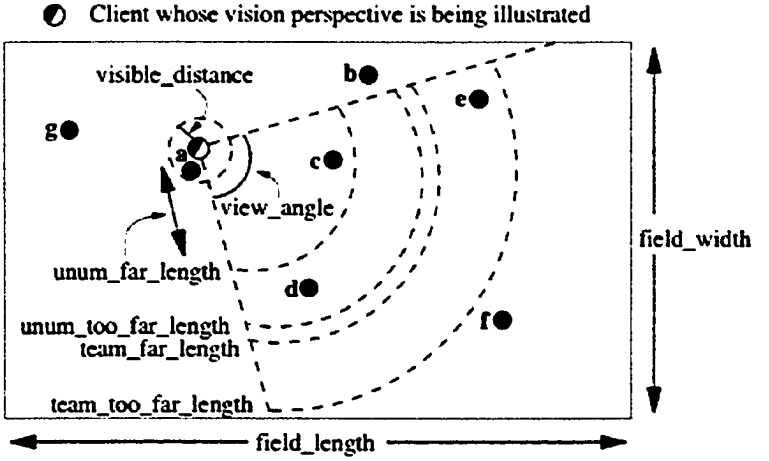


Figure 5.6: Objects around a RoboCup client, as seen from the clients perspective (diagram taken from the RoboCup Soccer Manual [10]).

5.4.4 Limitations

There are several potential problem areas to be aware of when considering object discretization [23]:

- introduction of bias
- edging and boundary issues
- overgeneralization

Bias

The raw RoboCup data provides many numerical attributes describing the observed objects, such as positions, distances, current velocities, velocity changes, etc. Any data representation that seeks to interpret and generalize this information — particularly without the added benefit of interpretation by a domain expert — introduces bias to the data.

For example, the allocation of objects to table cells relies only on the objects' position data. In reality, objects on the soccer field generally also have velocity vectors which may also be of importance. Design decisions that influence what data is kept and what is ignored, can result in data bias.

Boundary Issues

The artificial segmentation of the agent space into segments can result in a form of data bias as well as introduce issues at the boundaries between segments. Problems can occur if the recognition client assumes boundary values that are different from those used when originally generating the scene data. If the boundaries are not specified, an object classified into one region in one scene might be accidentally classified into an entirely different one in another, and the wrong match (and therefore wrong actions) could result.

The boundaries at which values are discretized must be chosen carefully to avoid issues with boundary values. Consider Figure 5.7, which illustrates two different types of boundary problems. First, *boundaries may be artificial.* Objects A and B in the figure are each classified into separate regions, since they are on either side of a boundary line. But due to their proximity to each other, they might just as

easily have been classified into the same region if the boundary lines had been placed differently. Which is correct? This artificial separation may impact search results during the scene recognition process. There may be a target scene which describes two objects together in one region. Although visually the depicted scene matches *such a description, the search algorithm might not match it.*

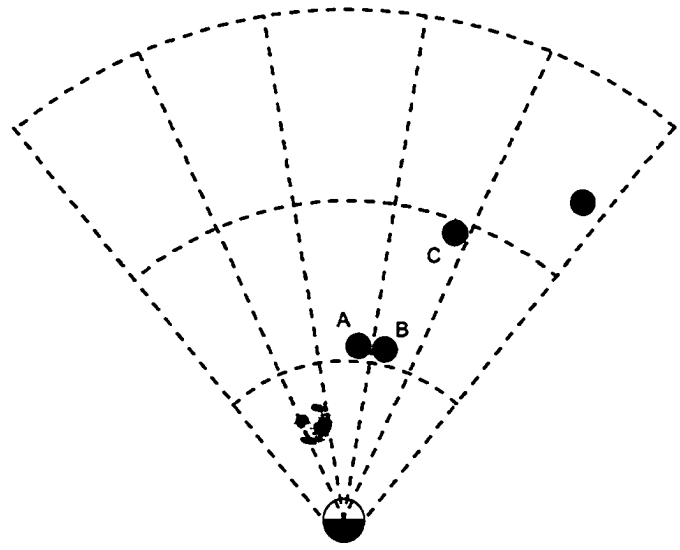


Figure 5.7: Boundary-edging issues in scenes.

Objects B and C illustrate *edging* and potential overgeneralization problems. Both objects fall into the same region in the scene. Visually, the objects are at extreme boundaries of their region. A strict interpretation of the region discretization suggests that B and C are closer to each other (i.e. in the same region) than A and B are (since A and B are in separate regions), although intuitively the opposite appears true. Consider also the potential semantic difference between an object that is well within a defined region versus an object that is within the region very close to the boundary.

Overgeneralization

Overgeneralization may occur as a result of discretizing two or more scenes that describe substantially different behaviours despite an overall similarity — one that may be emphasized once the objects in the scenes are processed into regions. Any scene recognition system must be able to analyze scenes that look similar and determine whether or not they actually are, otherwise the imitative algorithm will have several different choices to make without enough guidance on which to choose. But choosing too many parameters on which to base a decision may lead to *data overfitting.*

As an example, consider two scenes containing a teammate, the soccer ball and the goal, with both objects in similar positions in each scene (Figure 5.8). In one scene, the agent kicks the ball toward the goal. In another scene, the agent kicks the ball away from the goal. To an experienced soccer player, these two scenes clearly depict substantially different situations (one is a shot on the goal, another could be a result of a different objective, such as a pass).

Solutions

As with any generalization procedure, a balance must be maintained between how much information is discarded and how much is retained. One solution to the limitations shown above is to have discrete scenes also contain the original continuous-valued data. Algorithms could then use the original data for resolving any ambiguities. However, this also results in some duplication of data and increases the memory and storage requirements, and mitigates some of the advantages of generalization. However, by keeping both discrete- and continuous-valued data in the scene format, algorithms can be written to use either representation, even both, as appropriate.

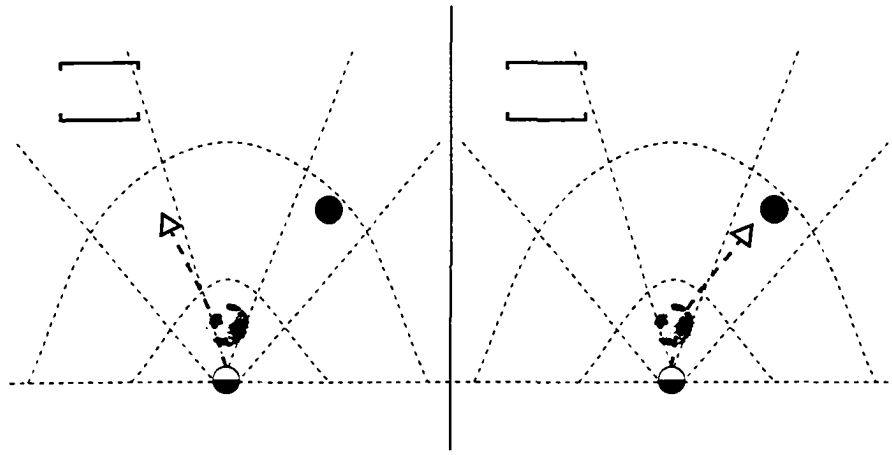


Figure 5.8: An example of data overgeneralization. These two scenes are different yet objects resolve to the same region-discretization.

To avoid boundary and edging problems, care should be taken not to choose boundary values arbitrarily. Better still would be to define boundary values at points that have semantic meaning, for example the server-defined distances (ball_kickable, etc.) This helps to assure that the region separation actually reflects a semantic difference.

5.5 Future Enhancements

The scene representation is flexible enough to support much more than simply being a static data representation. Scenes represent spatial information that can be manipulated in a variety of ways. The following sections explore some of these possibilities, which are largely outside the scope of this thesis and therefore left as future work. These include:

- operations on scenes

- optimization of scene coverage
- detection of key scenes
- scene symmetry
- scene memory
- future-scene projection

5.5.1 Scene Operations

As alluded to earlier and illustrated in Figure 5.4, operations can be defined to expose and manipulate the objects contained within scenes. Since a scene is a container for a set of objects, set manipulation functions could be defined such as unions, intersections, and differences between objects in scenes. These operations may be useful in providing other enhancements, such as adding scene memory: a superset of objects between consecutive scenes includes objects visible in one scene but not another, in effect remembering them from previous scenes.

5.5.2 Scene Coverage

Since each scene represents knowledge about one circumstance, there must be a sufficient number of distinct scenes to cover a wide variety of potential circumstances. The number of scenes must also be small enough to satisfy performance constraints (dependent on the algorithm complexity and the physical constraints of memory and CPU speed).

The most straightforward approach is to store every scene generated from the observation process. However, given that each observed (logged) player generates up

to 6,000 scenes, and scene collections may consider data from multiple players, this can quickly reach storage and performance limits.

Future work may consider clustering or elimination of similar scenes. Scenes in which similar sets of objects are classified into the same regions — and (importantly!) the resultant actions are equivalent — may be similar enough to condense together. Such a process of generalization may reduce the number of scenes required to search through during the recognition process.

5.5.3 Key Scene Extraction

When humans learn by observation, it is intuitive to watch, for example, a replay of a soccer game and digest the entire replay into several key times representing key events (major passes, blocks, etc.) Additionally, attention tends to be focused on specific objects in the field; the remaining objects are considered irrelevant.

Out of the many scenes describing a particular game, there may only be a small subset which represent truly remarkable behaviour. Additionally, within these "key" scenes, only some of the objects are of relevance. Event and feature extraction algorithms could be applied to determine which scenes represent important events worth noting and which objects are involved in these events.

Another factor to consider is whether one scene should necessarily only represent information from one time unit in the observed logs. Scene algorithms could be used to extract and represent knowledge from multiple times (typically time just preceding and following the main events being observed).

5.5.4 Scene Symmetry

Another approach to help increase coverage while maintaining a minimum number of stored scenes is to consider scene symmetry. Figure 5.3 illustrates two scenes which are identical except for a reflection along the y-axis, i.e. mirror images. If matching algorithms take advantage of the symmetric nature of most scenes, reflected scenes could be eliminated for a significant reduction in the number of stored scenes required at the same level of coverage. This will result in better accuracy since coverage can then be increased to include more diverse scenes.

5.5.5 Scene Memory

As an agent travels through the soccer field, some objects come into view and other objects disappear from view. An object visible at time t may be just out of visual range at time $t+1$. The current position of the object may be extrapolated from the velocity vectors provided in the last scene in which the object was still visible. Scenes could include a built-in memory that retains such information. This would allow the agent to remember, for example, that the ball was recently nearby and should still be nearby (if not visible, perhaps behind the agent). As time continues forward, the confidence of such a memory would decrease accordingly.

Scenes in this case would not be considered as standalone, but in groups by time. Objects seen at time ($t-1$, $t-2$, etc.) that are no longer visible at time t would be carried over as "remembered" objects in the current scene.

5.5.6 Scene Projection

In addition to considering previously-seen objects and extrapolating where they might currently be, scenes might also be used to project forward in time and predict the location of objects in the future. Since most objects have velocity vectors associated with them, scenes can be used to anticipate the future to some degree. This may be handy, for example, in an algorithm designed to predict the future location of the ball and move to intercept.

5.6 Conclusions

This chapter introduced a scene-oriented representation of data captured from RoboCup simulated soccer games. Scenes are largely conceptual and provide a convenient graphical representation which is intuitive to visualize. Introducing region discretization provides a level of generalization and suggests that there are semantic meanings attached to regions of values (e.g. particular ranges of distances representing a space that is classified as "nearby" or "far away"). Scenes provide a description of objects that are relative to the player (observer) and not to the absolute coordinate system of the soccer field.

The scene definition implies a number of possible manipulations of scenes, some of which have already been addressed. Scenes may be manipulated in the following ways:

- Set operations including unions, differences and intersections
- Scene reflections along the horizontal axis (symmetry)

- Matching of one or more scenes to another scene (Chapter 6 is devoted to this topic)
- Human visualization and manipulating of scenes (e.g. using a GUI)

The scene concept is implementation-independent (Chapter 7 describes a lightweight implementation of scenes as an enhancement to an existing RoboCup agent's knowledge representation). It suggests a range of potential uses from simply being a data representation (recording the positions of objects during a game, such as in a log-playback tool) to being a representation suitable for expressing high-level tactics (such as a locker room white board used by a coach to describe plays to his team).

The key to the effective use of the scene representation will be the algorithms for manipulation and matching of scenes, which are discussed in the following chapter.

Chapter 6

Scene Recognition

> It does not take much strength to do things, but it requires great strength to decide on what to do.
>
> *Elbert Hubbard*

The scene description format detailed in the previous chapter is a lightweight representation that supports a variety of uses and operations. Scenes can be used both in low-level tasks, such as annotating the raw data, through higher-level tasks, such as describing play sequences or tactics.

The scene format itself is straightforward. Its primary strength is in the potential for manipulating the data in new ways thanks to the structure and additional semantic meaning introduced by the scene format. Two primary objectives may be achieved using the scene representation:

- generalization, discovering patterns, data mining, or other higher-level learning
- use in a scene-recognition algorithm suitable for real-time deployment in a RoboCup client

The first goal is largely beyond the scope of this thesis (though we do provide possible directions and suggestions based on experimental results — this is described in detail in Section 9.3). This chapter describes the latter goal, the process of developing a behaviour recognition algorithm for use with the scene format.

6.1 The Recognition Problem

The scene recognition approach is used to help a new software agent answer the question, *"what should I do in this situation?"* by considering the behaviour of a previously observed agent. That is, given a new situation $N = \{x_1, x_2, x_3, \cdots, x_n\}$ (recall each x_i represents an object observed on the soccer field), what is the most appropriate action to take?

We approach the problem given a set of scenes $S = \{S_0, S_1, \cdots, S_n\}$ (tuples of observations and related actions) describing the original agent's behaviour. Recall that each scene contains a set of inputs I and the corresponding reaction R. If any scene $S_i = N$, then the problem is solved — the recorded action R can be reused. (Note that the new agent may choose to use this information as knowledge to help derive a new decision, or simply reissue the same command. The latter must take into account the specific parameters associated with commands — how far to kick, and in what direction, for example. See Section 6.3.)

Realistically, given the dynamic and random nature of the RoboCup domain, it is improbable that any stored scene will exactly match a given situation. To expect such would require an infinitely large set of stored scenes. Thus, the problem is to find the scene S_x which is "closest" in configuration to the situation N.

6.1.1 Scene Recognition Algorithm

The most straightforward approach would be to use a 1-nearest-neighbor search to select the single "closest" match between the current situation (events unfolding in the actual game) and the collection of stored scenes. The client can then trigger an action that is based on the action associated with the matched scene. The general case uses k-nearest-neighbor matching; the k best scene matches and corresponding actions are all considered, then evaluated using a separate action selection algorithm to decide which action will ultimately be chosen.

The scene recognition process follows the general algorithm shown below:

```
load the stored scenes from disk
while(game on):
     get new incoming ''see'' data from server
     convert data to scene object N
     for each stored scene x:
          d = DistanceCalculation(N, x)
          keep k best results
     end loop
     for the set S of k best candidate scenes:
          a = ActionSelection(S)
          keep best recorded action a'
     end loop
     generate new action a'' based on a'
     send action a'' to server
end loop
```

6.1.2 Constraints

The simulation algorithm must consider the constraints placed on its performance due to the real-time nature of the soccer server environment. In such an environment,

time and *memory* may both be at a premium.

Two factors will impact the performance of the search algorithm: the algorithm's complexity, and the total number of scenes to search. The algorithm complexity impacts execution time. The stored scene size also impacts a number of other factors:

- search algorithm performance (execution time)
- memory and disk storage
- start-up time (loading the scene set into memory)

The algorithms should be designed as much as possible to minimize the computation time. The required size of the scene collection (coverage) will likely be determined experimentally and depends on the behaviour of the agent being observed (and may be further optimized using generalization or pruning techniques).

6.2 Distance Calculation

We now consider the `DistanceCalculation(N, x)` function defined in the scene recognition algorithm. This function is used to determine a value of "distance" between two scenes; this distance is then used to determine which scene is "closest" to the target.

Consider first the distance calculation between two single points with Cartesian coordinates (x_1, y_1) and (x_2, y_2). This is a straightforward calculation:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{6.1}$$

Since scenes are collections of multiple objects in space, the distance between two

scenes is the sum of the distances between the objects within the scenes. Essentially, the distance calculation algorithm must define a mapping between objects in one scene and objects in another, and determine individual distances between objects, to perform its calculation:

Definition 1 *Given a scene S_1 with objects $\{x_1, x_2, \cdots, x_n\}$ and a scene S_2 with objects $\{y_1, y_2, \cdots, y_n\}$ the distance d is defined as*

$$d(S_1, S_2) = \sqrt{\sum (x_n - y_n)^2} \tag{6.2}$$

(For the moment, we assume homogenous object types. Object classification by type is discussed in Section 6.2.3.)

6.2.1 Continuous distance calculations

In the simplest case, the distance and direction values provided by the soccer server can be used to directly perform the distance calculation. Objects on the field are all described with a distance and a direction relative to the observing agent. This can be represented by a polar coordinate $P(r, \theta)$, with the agent located at the origin. Therefore, distances between objects and the client, or distances between two objects, can readily be calculated using the cosine law (Figure 6.1).

Individual object distances, whether in Cartesian or Polar form, are straightforward. Calculating the total distance between groups of objects in two scenes becomes more difficult, since the algorithm must first optimally match objects from one scene with corresponding objects from another.

Consider the example in Figure 6.2, which shows two scenes A and B each with

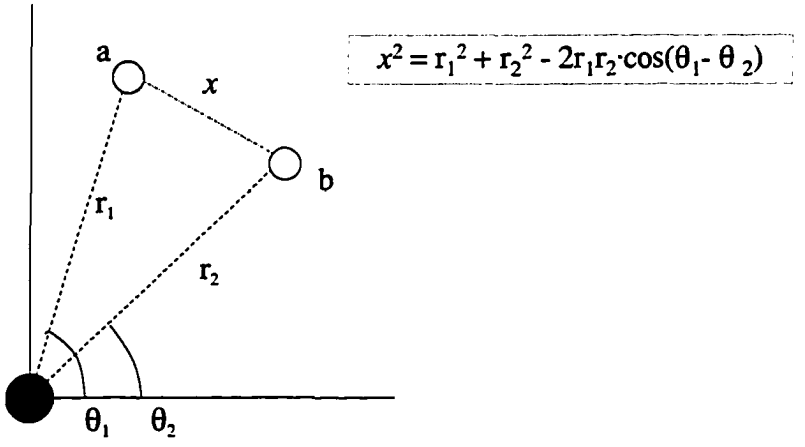


Figure 6.1: The cosine law is used to find the distance between objects in two scenes.

two objects. One can intuitively observe that these two scenes appear similar (e.g. compared to the very different looking scene in Figure 5.2). This can be established since the two objects in each scene are "similarly" positioned — object a can be matched with object c and object b with object d. In fact, if $dist(a, c) = 0$ and $dist(b, d) = 0$ then scene A = scene B, that is the two scenes are a perfect match. Otherwise, the *total distance* between scene A and scene B is the sum

$$d = (dist(a, c) + dist(b, d))$$

(Based on equation 6.2). As the distances between matching objects decrease, the distance score also decreases, converging upon a perfect match when the *total distance*, $d = 0$.

An algorithmic approach to object matching is a major factor in designing the

distance calculation algorithm and is discussed separately in Section 6.2.4.

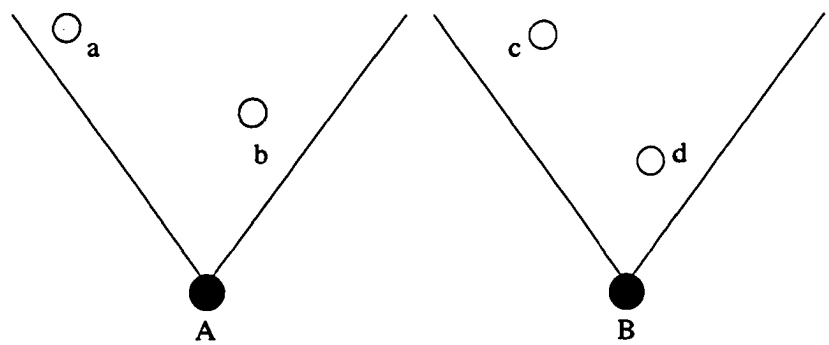


Figure 6.2: Two closely-matching scenes.

6.2.2 Discretized distance calculations

It is now possible to see the potential advantages of scene discretization. Consider again the two scenes shown in Figure 6.2. Intuitively, scene A and scene B look very similar, though a precise distance calculation would show otherwise. In fact, when the discretization process is applied and the objects in these scenes are partitioned into the appropriate "cells", objects a and c might be placed into the same cell, as would objects b and d. A distance calculation based on cellular regions rather than precise object locations would thus declare $A = B$.

This can be advantageous because equivalent scenes do not all need to be stored[1]. Pruning the scene store based on these similarities can reduce the size of the stored

[1]Note that if scenes A and B each have a different corresponding action, they might not be considered equivalent!

scene set, which also results in a time optimization. Alternately, more diverse scenes can be stored resulting in greater scene coverage.

In the discrete-region form, object positions are no longer represented by their position $P(r, \theta)$ but by the (x, y) grid coordinates of the region in which the object lies (also equivalent to the column and row positions of the cell when represented as a table, Figure 5.1).

For example, if object a is in cell $(3, 3)$ and object c is in cell $(2, 3)$ then the distance between them, using the Cartesian coordinate distance between cells, is 1 (equation 6.1).

6.2.3 Object Correspondence by Type

Until now we have assumed homogenous object types, that is, we have freely supposed that objects in a scene can be matched with any objects from another scene. In fact, this is not the case — the matching of objects is complicated by the fact that there are actually numerous different *types* of objects on the soccer field. Even within type classes, there are subtypes; for example, there are different types of players (teammates and opponents) and goals (ours and theirs), and flags and lines carry unique identifying markers (e.g. the 30-yard line on the right top side).

Intuitively, only like objects should be matched with each other; it is meaningless to compute the distance between a ball in one scene and a player in another.

To solve this problem, we first break down the set of objects in a scene into separate sets of object types, then calculate the distance d between two scenes as the

sum of the individual distances between different object types:

$$d_{total} = d_{ball} + d_{teammates} + d_{opponents} + d_{our-goal} + d_{their-goal} + d_{flags} + d_{lines} \quad (6.3)$$

Some objects, such as the ball, are easily matched: by definition, the ball in one scene must be matched with the ball in another. In cases where there are multiple objects of a given type (e.g. players), the players in one scene must each be matched with the equivalent players in another.

Objects with multiple subtypes, such as players, lines and flags, can be further sorted by their subtypes, or for simplicity and to increase generality, they may simply be grouped with other objects of the same general type. This increases generality and makes the matching algorithm much simpler (and therefore faster), but at the cost of decreased accuracy. Matching individual subtypes increases accuracy, but risks data overfitting. The point where the trade-off becomes acceptable is another design consideration.

6.2.4 Object Matching

Previous sections have alluded to the somewhat nebulous problem of object matching between scenes. The individual distance algorithms — whether in continuous or discrete form, using polar or Cartesian coordinates — all operate on pairs of objects, one from each of the two scenes being compared. At this level, distance calculations are straightforward — the true complexity of the distance calculation algorithm actually lies in determining the mapping between these pairs of objects.

In fact, this is an example of an optimal object-pairing problem, where the cost function to be minimized is the distance between each pair of objects. Depending

on whether the two scenes have equal numbers of each object type, it is classified as a minimum-weight bipartite matching or minimum-weight non-bipartite matching problem. A popular solution to this problem is Edmonds' blossom-shrinking algorithm [15, 53] which is known to run in polynomial time. A simpler approach would be an application of the *skiers and skis* matching problem [50, 15] (an optimal solution exists only when both lists are of equal size).

Because of the complexity of implementing the optimal matching algorithm, a simple, rough heuristic based on the skiers and skis problem can be used. The objects in each scene can be sorted by their distance to a common reference point (e.g. the player itself). Consider scene A with a sorted list of objects $(a_1, a_2, \cdots, a_n)$, and scene B with objects $(b_1, b_2, \cdots, b_n)$. (For simplicity, homogenous objects are assumed; this process is actually repeated for every object type.) The objects can then be paired by $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$; the total distance is the sum of each $dist(a_n, b_n)$.

While this algorithm is simple, its weakness lies in the fact that it is essentially one-dimensional. That is, simply because object a_i and b_i have similar distances from the common reference point, does not necessarily suggest that the two points are in close proximity; they may simply be concyclic. Instead, it is treated as a rather rough heuristic. Implementation of an optimal solution, such as Edmond's algorithm [49], would improve the quality of the object matchings (but due to complexity and time constraints, are left as future work).

Non-Bipartite Matching Considerations

In general, it is likely that two scenes A and B will not contain perfect correspondences between sets of objects. One scene may contain more flags, than another, for example,

or more players.

In this case, a distance calculation may be made based on the common subset of objects between the two scenes, but the "missing" objects must also be taken into consideration. An object that is present in one scene but totally absent in the other clearly cannot be a match. However, since the common subset is the empty set $\emptyset$, and by definition $dist(\emptyset) = 0$, special consideration must be taken to ensure that empty object sets do not inadvertently get misinterpreted as perfect matches.

One approach is to artificially increase the distance value by adding a "penalty" distance for objects in one set that have no corresponding object in the other. This penalty should be proportional to the distance — if scenes A and B only differ by one object that is far away from the player, the penalty should be lower than if the object is very close. Intuitively, this is a measure of the importance of an object, though this is still just an approximation with limitations (e.g. if an agent is making a long goal shot, the existence of the goal in the scene — even if very far away — is certainly important!)

6.2.5 Object Weighting

Finally, we consider the relative value of different types of objects. Scenes may contain balls, goals, teammate and opponent (and unknown!) players, flags, and lines. Each object type may be given a weight w. By default all objects are considered equally, i.e. $w_{ball} = w_{opponent} = w_{teammate} = 1$. It may be desirable to not consider certain types of objects at all (i.e. lines on the field may be irrelevant to the scene), in which case the weights can be set to 0.

Thus, the distance calculation is

$$d_{total} = w_{ball} * d_{ball} + w_{teammates} * d_{teammates} + w_{opponents} * d_{opponents} + w_{our-goal} * d_{our-goal} + w_{their-goal} * d_{their-goal} + w_{lines} * d_{lines} + w_{flags} * d_{flags} \quad (6.4)$$

$$d_{total} = w_{ball} * d_{ball} + w_{teammates} * d_{teammates} + w_{opponents} * d_{opponents} + w_{our-goal} * d_{our-goal} + w_{their-goal} * d_{their-goal} + w_{lines} * d_{lines} + w_{flags} * d_{flags} \quad (6.4)$$

Or, to expand on equation 6.4, the total distance calculation may be expressed as

$$\begin{aligned} d(S_1, S_2) &= \sum_i \left[w_i * \left(\sqrt{\sum (x_{in} - y_{in})^2} \right) \right] \quad (6.5) \\ S_1 &= \{x_1, x_2, \cdots, x_n\}, \\ S_2 &= \{y_1, y_2, \cdots, y_n\}, \\ i &\in \{objects\} \end{aligned}$$

Weights may be derived experimentally or from knowledge of the domain.

6.3 Action Selection

Consider the second function in the scene selection algorithm, `ActionSelection(S)`. This function filters the results from the `DistanceCalculation(N, x)` algorithm and is what determines which action the agent ultimately chooses to execute.

The general case scene recognition algorithm uses a k-nearest-neighbor approach in determining the "closest match" between the key scene and the set of stored scenes. The result of the search is a collection of k scenes, each of which is (by definition) a potential match. Each of these scenes is associated with an action (Table 2.2),

typically one of the three most common actions (*dash, kick, turn*)[2]. Additionally, each action has associated parameters (power, direction, etc). Thus, even narrowing down the selection to k similar scenes still leaves numerous possible actions to consider (and even similar action types may have different parameters, such as two kicks with different directions).

The agent may choose to do one of the following based on the available information:

- Choose one of the prerecorded actions to follow exactly,
- Using the prerecorded actions as guidance only, generate a new action accordingly,
- Generate actions based on the agent's own domain knowledge.
- Some combination of the above.

Since the latter choices are implementation-dependent, we consider some strategies for choosing actions based directly on the information from the prerecorded scenes:

- pick one action at random from among the candidate scenes (typically for statistical and testing purposes only)
- pick the first valid action (protect against corrupted or invalid data)
- pick the most common action (majority vote)
- some combination of the above

[2]Other actions are possible, such as *catch* (a goalie-only action), or *turn_neck* (may or may not be ignored, depending on implementation). Other commands are also possible (e.g. request to resend status) but these have no impact on the physical world and are thus not considered.

Attribute values such as power and direction may also be taken into consideration during the selection process, or they may be analyzed separately (Section 6.3.2).

6.3.1 Action Weighting

The RoboCup domain allows an agent to take various actions which may or may not impact the environment (e.g. *kick* versus *turn_neck*). Whether or not an action such as *turn_neck* should be executed or ignored may depend on the agent implementation and experimental results. The consequence of issuing a *turn_neck* command is a change in the perspective of the incoming sensory data, with no direct consequence on the surrounding objects. The imitative agent should be prepared to track the neck angle and offset measurements accordingly.

To address this, different actions can be classified with different weights, effectively either ranking their priority or allowing (when $w \in 0, 1$) them to be disabled. Thus, the typical action selection algorithm becomes a weighted majority vote.

6.3.2 Action Parameters

Finally, once an action is chosen (typically to dash, to turn, or to kick), the parameters associated with that action must also be chosen. For example, if the action chosen is *kick*, the agent must decide which direction to kick in and at what strength. The simplest approach would be to just reuse the values originally stored in the matching scene. By definition, since the target and stored scenes are similar, it is likely that the target of the kick is in a position similar to the target of the stored scene; in this case, a similar direction and power should achieve similar results.

It is possible this may not be the case, and the target of the kick might be one

of few objects which do not match closely between the key and stored scenes. One approach is to consider the average of the parameters when using a majority vote algorithm; if there are n different instances of a *kick*, the average kick power and direction could be taken over the n results.

Ideally, the imitative agent should use the stored parameters as a guide and not simply copy the stored values. One logical approach to do so would be for the agent to attempt to interpret the stored scene — i.e. was the {*kick, turn, dash*} aimed at a particular object? This could be determined by examining objects located in the scene within the direction and range of the action. For example, if a kick was issued in the direction 33°, and the scene contains a goal located in the same direction, it can reasonably be assumed that the intent of the agent was to kick the ball into the goal. This may not always be so straightforward to detect, since the agent may have taken other dynamics into account (such as kicking or dashing along an intercept vector aimed at another moving object). At least a few degrees of error should also be allowed to account for noise in the RoboCup data.

6.4 Conclusions

The proper scene recognition algorithm is crucial to the success of an imitative agent. Algorithms are subject to real-time constraints placed by the environment, including limited execution time, storage, memory, and data availability (and also accuracy, in the case of an environment like RoboCup where noise is present in the data).

The algorithms presented in this chapter are relatively straightforward and suitable for a simple recognition system subject to the assumptions made in Section 4.2. That is, the client is assumed to be a single-layered client architecture (Section

3.1.2) in which the inputs directly correspond to the outputs, without internal states or memory of previous events or actions. While this may at first be an oversimplification, it is a suitable starting point upon which further enhancements, such as generalizations, states, memory, etc. may be built.

The two core problems faced by the scene recognition agent, when confronted with a new environment, are determining what similar situations the recorded agent may have encountered before, and what actions were taken at that time. The `DistanceCalculation()` and `ActionSelection()` algorithms address each of these issues; within each algorithm there is still plenty of room for enhancements and optimizations.

These algorithms support much tweaking through the use of varying object weights or use of different selection methods (calculation using discrete distances, for example, versus continuous-form data). Many of these tweaks can be made manually or under a (possibly automated) testbed, remaining true to the original goals of developing a hands-free training process. Key to this will be the development of an objective measurement of success which will help determine which algorithms, and which parameters, to use for a given imitation problem.

The following chapter describes an implementation of the scene format and algorithm built around an existing RoboCup agent.

Chapter 7

Implementation

> For the things we have to learn before we can do them, we learn by doing them.
>
> *Aristotle*

Chapters 5 and 6 describe a scene knowledge representation format and an algorithm for manipulating scenes and performing a form of simple scene recognition. This chapter describes the implementation of the scene format and the scene algorithms, including both a framework for generating and working with scenes as well as a complete real-time scene recognition system built into a RoboCup soccer agent.

In addition to the scene manipulation framework, this chapter also provides some implementation details concerning the other steps of the methodology (Chapter 4), including the initial log capture process and scene generation (conversion) from the captured log files.

7.1 Design of a Scene-Based Agent

Since the eventual objective is to develop a RoboCup soccer agent which applies the scene-recognition algorithm for imitative learning, the bulk of the development revolves around building the RoboCup agent. Beginning with a basic agent framework, we develop the following:

- knowledge representation using an implementation of scenes
- a decision-making framework using the scene recognition algorithms (distance calculation and action selection)
- specific implementations of various calculation algorithms

7.1.1 General Design

The block diagram of Figure 7.1 illustrates the components (and part of the data flow) for a scene-recognition-based RoboCup agent.

As with any RoboCup client, the process begins with the influx of visual sensor data from the soccer server, which completely describes the state of the environment at a specific time. In order to be compared to the stored set of scenes, this incoming visual data must itself be converted into a scene. For simplicity and consistency, the code that performs this function in the real-time agent should be the same code that converts logs into scenes (Section 7.4).

Once the "current" environment state has been encoded as a scene, it can be compared with the set of stored scenes using the recognition algorithms. The stored scenes serve as the agent's knowledge base, ideally covering as many situations as

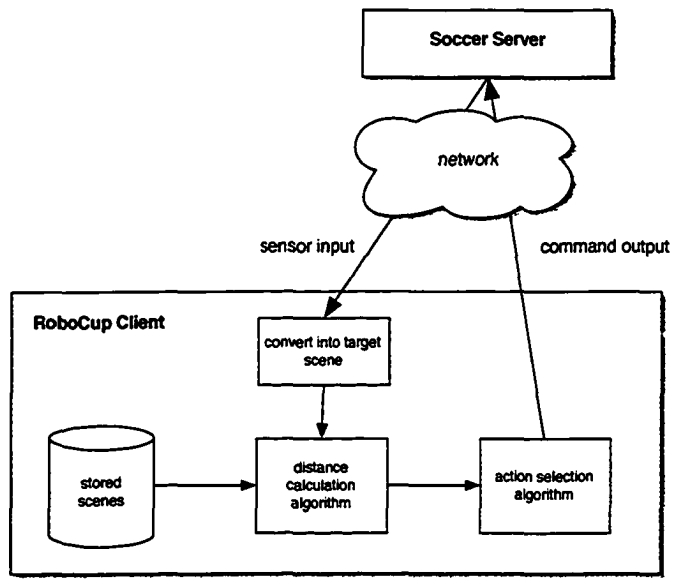


Figure 7.1: Block diagram of the general architecture of a scene-based RoboCup client.

possible. The agent's implementation of the DistanceCalculation() and ActionSelection() algorithms are what drive its decision-making. The chosen command is sent to the server for execution, and the cycle repeats.

7.1.2 Performance Constraints

The entire decision process must be fast enough to happen within the time of one RoboCup soccer server simulation cycle. A typical game consists of 6000 clock cycles, which occur at a rate of 10 cycles per second. Thus, a soccer client has about 100 ms to make a decision and send an action command back to the server. If a client uses time beyond this 100 ms, its performance on the field will appear to lag other players since cycles may be skipped before commands are sent. The length of a clock cycle may be changed in the server configuration file, and there does exist a synchronous

connection mode, if a longer time between cycles is required.

7.2 Modifying the Krislet Agent

In order to facilitate code reuse and reduce development time, an existing RoboCup client was used as a starting point. The Krislet client [40] was chosen for the following reasons:

- Coded in Java for portability, ease of development and debugging
- All source code is readily available[1]
- Code is well-documented, easy to understand and modify
- Object oriented approach provides data structures for RoboCup objects
- All client-sever communication and message parsing is already implemented

The following sections provide some background on the Krislet client's implementation as well as the changes made to the base code to support scene recognition.

7.2.1 Krislet Architecture

Kristof Langner's Java-based Krislet client was described in Section 3.1.1. Developed as a prototype for an open and extensible agent architecture (the "Brain" framework), it is an ideal starting point since it already encapsulates all of the more difficult aspects of RoboCup agent programming including the UDP client-server communication protocol and message decoding and parsing. The Krislet code in fact provides a complete

[1]This is important, since most RoboCup agent research revolves around teams that implement proprietary strategies to which the full source code is not released.

set of objects corresponding to RoboCup soccer objects (e.g. players, flags), which greatly simplifies the implementation of the scene format (Section 5.2).

Figure 7.2 is a partial class diagram which shows the basic architecture of the Krislet client. For simplicity, some utility classes and interfaces, such as the `SoccerParams` class (a storage class for various parameters and state variables) and the `SendCommand` interface (for the client-server communications), are not shown. Some class members and methods are not shown in the diagram..

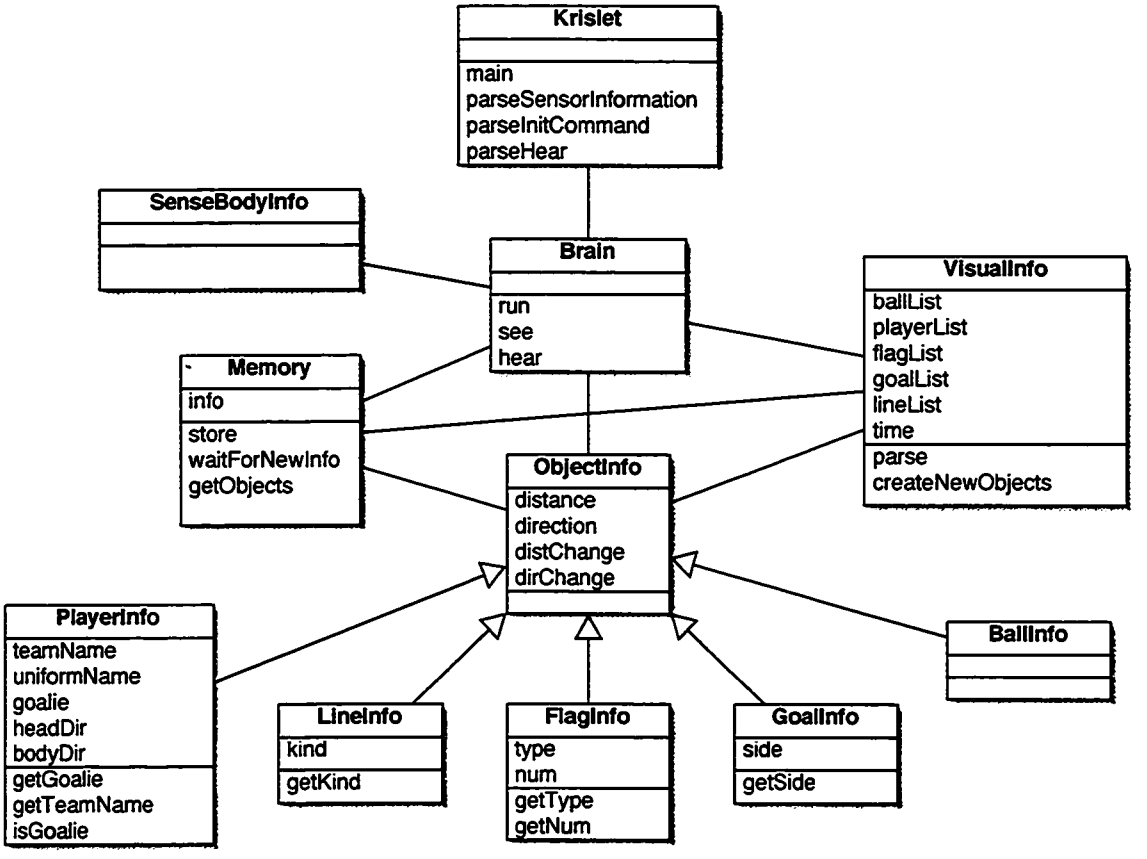


Figure 7.2: A partial class diagram showing the architecture of the Krislet client.

The agent is instantiated by the `main()` method in the `Krislet` class. This class

defines a set of methods for connecting to the RoboCup soccer server and methods for sending control primitives (dash, turn, kick, etc.) as commands.

The `Brain` class is the aptly-named center of control for the operation of the Krislet agent. Most of the logic of the agent (including the default set of decision rules described in Section 3.1.1) is defined in this class. A `Brain` object is instantiated and runs in its own thread. When new messages arrive from the server, the data string is placed in the `Memory` storage class for the `Brain` to process.

The `Brain` thread simply polls for updates of the `Memory` object, and when it arrives, parses the incoming data stream. Messages can be one of `see`, `sense_body`, or `hear` (these are the three possible message types sent from the server; see Table 2.1).

These three message types are each handled separately. In its current implementation, `hear` messages are ignored. All `sense_body` messages are parsed into a `SenseBodyInfo` object which updates attributes representing each value. (Krislet does not currently make use of this information.) Similarly, `see` information is parsed by the `VisualInfo` class.

The `VisualInfo` class is both a parser for `see` messages as well as a container for RoboCup objects. One instance of a `VisualInfo` object represents everything described in a given `see` message. Each type of object visible on the soccer field — players, balls, goals, lines and flags — has an associated class with appropriate attributes including perceived speed and direction. These classes (`BallInfo`, `GoalInfo`, etc.) all inherit from a common superclass `ObjectInfo`. The subclasses specialize as appropriate — for example, players have teams and uniform number attributes associated with them. A `VisualInfo` object effectively is a container for `ObjectInfo`

objects.

7.2.2 Adding Scene Support to Krislet

Given that a scene is, conceptually, simply a collection of objects from a given point in time, it seems natural to exploit the functionality already built into the `VisualInfo` and `ObjectInfo` classes. In fact, only minor changes are necessary in order to support spatial discretization parameters (described in Section 5.4); much of the existing code is simply reused.

To support the discretization of space into regions, we need only to define additional attributes in `ObjectInfo` to identify which cell coordinates, in the form $(column, row)$, each object would appear in. The coordinates represent the column and row numbers of a table which would describe the scene (see example, Table 5.1). Existing object attributes (e.g. distance, direction, team name) remain unchanged (these are made available for those distance calculation algorithms that do not use object region discretization).

A new `Action` class is defined which enumerates the available RoboCup actions (e.g. *ACTION_DASH, ACTION_TURN*). Finally, a new `Scene` class is defined, containing a reference to the modified `VisualInfo` class and a list of `Action` class references. Each `Scene` object thus contains a set of visible RoboCup objects and the corresponding set of actions. The Scene object in effect describes a cause-effect relationship: when these objects are found, the corresponding action is taken. An artificial limit of 2 actions per scene prevents scenes from containing more actions than strictly necessary (based on observations that many of the captured log files contained what appeared to be two actions per given time index). The RoboCup server only allows

one "world-modifying" action to be executed per time index. The `Scene` object also provides useful attributes including an identification string and parameterizable table size (for discretization). It also includes a console-based visualization (a `toString` method) for convenience.

7.2.3 Scene Recognition Classes

A version of Krislet was developed, dubbed *KrisletScenes*, which adds new scene-related classes and incorporates minor modifications to existing Krislet classes to accommodate the extra data associated with scenes. Figure 7.3 shows the relationship between the new classes and some of the existing Krislet classes (for simplicity, only relevant classes are shown, and test classes are omitted).

A number of other new classes were developed to create a scene-recognition framework, which are briefly outlined below:

- The `LogToScenes` class is a standalone utility which parses a captured log file containing client-server messages, converts it into a sequence of scenes and saves it to a disk file. (Section 7.4 details this class further.)

- The `SceneSelection` class contains a set of methods implementing various action selection algorithms, including a weighted majority vote, random selection, etc. (Section 6.3)

- The `DistanceCalculation` interface defines method signatures for a distance calculation algorithm (Section 6.2) to determine the distance between two scenes and to determine the best match between a key scene and a list of candidates.

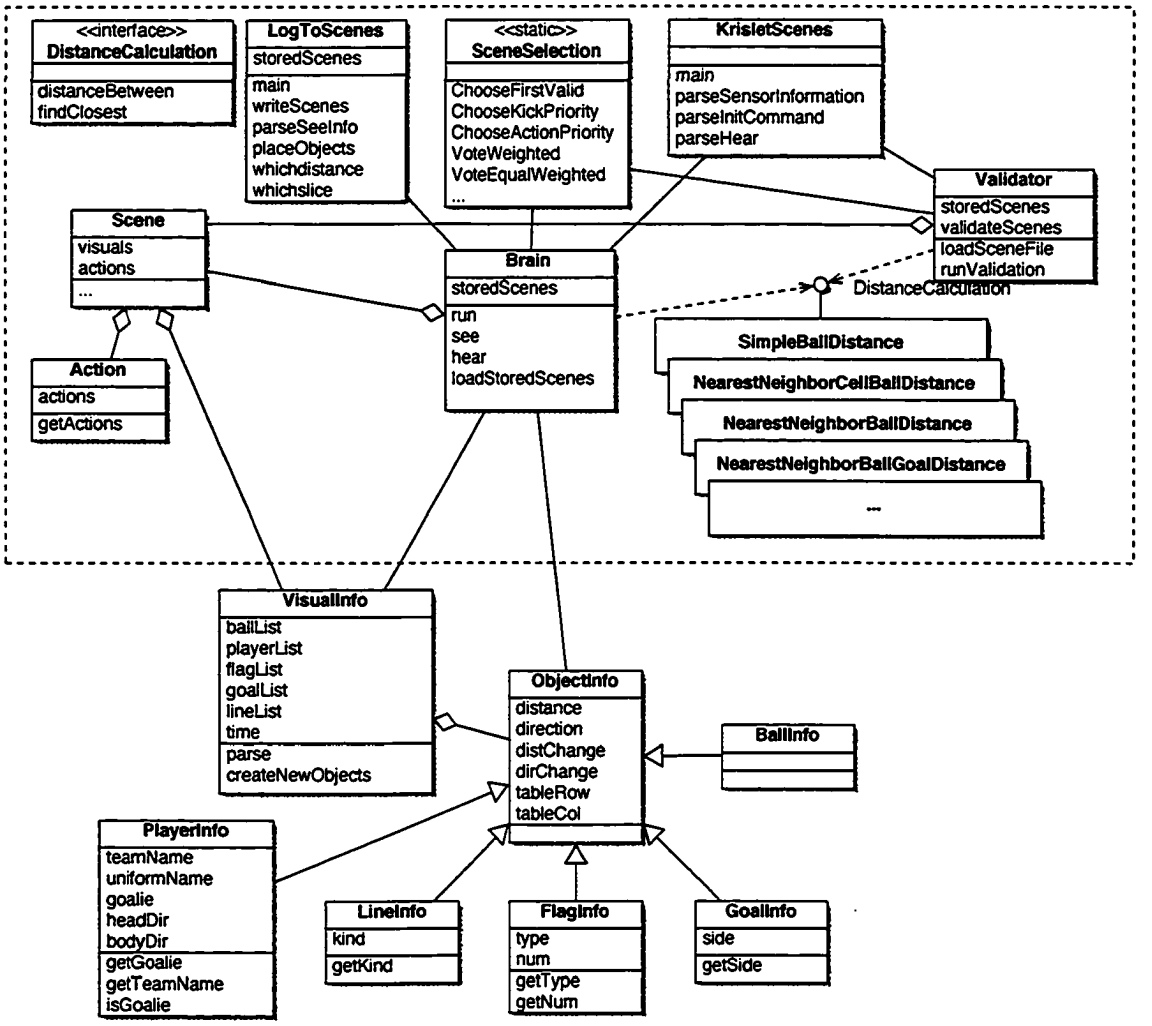


Figure 7.3: A partial class diagram of the KrisletScenes client; new classes are enclosed in the dashed outline.

Different approaches and algorithms can be used, each implementing this interface, so that different algorithms can be "plugged in". The interface also supports setting different object weights (Section 6.2.5).

Specific implementations of the `DistanceCalculation` interface are discussed separately in Section 7.3.

In addition to the new classes, the following changes were made to existing Krislet code:

- `VisualInfo` was refactored and made `Serializable` so that instances may be saved and restored from a disk file.
- Two new members were added to `ObjectInfo` to identify the row and column of the "virtual" cell in which the object would be located when the scene is discretized.
- The `run()` method of the `Brain` thread was modified to support a new scene recognition loop rather than the default Krislet decision rules.
- The start-up code parses new command-line parameters and loads a stored set of scenes from a disk file before starting the `Brain` thread.

7.2.4 KrisletScenes Execution Logic

The original Krislet's execution was contained in a loop which awaited messages from the server and followed a set of simple hard-coded decision rules (Section 3.1.1). KrisletScenes operates in a similar manner, though the decision logic has been modified to use the scene recognition algorithms. The following steps describe the operation of KrisletScenes:

1. On startup, a set of stored scenes is read from disk and stored in memory.
2. An object implementing the `DistanceCalculation` algorithm is instantiated.
3. The `Brain` thread waits for new server messages. When new visual data is received, it is parsed into objects and stored in a `VisualInfo` object (unchanged from the default Krislet implementation)
4. The `VisualInfo` object is passed through a static method in `LogToScenes` which calculates the discretized-scene coordinates of each of the objects. A new `Scene` is created containing the `VisualInfo` object. (This step could be skipped if the distance calculation algorithm is known to ignore the discretized data; for simplicity, it is currently always calculated.)
5. The `findClosestMatch()` method in the `DistanceCalculation` object is called with references to the stored scene set and the new `Scene`. It returns a subset of potentially matching scenes.
6. The scene subset is passed to a `SceneSelection` method, which returns one definitive action to take.
7. The `Brain` thread sends the appropriate action command is sent back to the server.

7.3 Algorithm Implementations

An important design decision is the ability for new distance calculation and action selection algorithms to be created and used with KrisletScenes or with any other scene-based agent. The `DistanceCalculation` interface allows algorithms to potentially

take any form, including use of object methods, static methods, separately running threads, even interfaces with other tools, such as WEKA, or a GUI interface.

A design decision was made not to use an interface for `SceneSelection` but instead to make use of one class with many static methods. This was for two reasons: varying (and unpredictable) method signatures between approaches, and a high degree of inter-operation between methods. For example, a `ChooseFirstValid()` method requires no parameters, while a `VoteWeighted()` method requires method weights. Some methods are called hierarchically, for example the implementation of most action selection methods actually make use the `ChooseFirstValid()` method.

The following sections describe the implementations made for the various distance calculation and action selection methods.

7.3.1 Distance Calculation Algorithms

Section 6.2.1 and Section 6.2.2 describe two general categories of the distance calculation, using continuous-form and discretized-form measurements of object positions. An implementation of each type was produced, each using a k-nearest neighbor approach. These include support for attributing different weights to different object classes.

Additionally, for statistical and testing purposes, a "random" distance calculation was created which does not consider distance measurements at all. This calculation simply picks any scene at random, and is used only to establish a statistical baseline to help grade the performance of other distance calculation algorithms.

Finally, a specialized distance calculation (NearestNeighborCellBallGoalDistance) was created which considers just the ball and the opponent goal. This method was

created by a group of graduate students who used the KrisletScenes framework as a course project (see Section C for details).

A total of four `DistanceCalculation` implementations were created, as described in Table 7.1.

Table 7.1: DistanceCalculation Algorithms.

RandomDistance	Chooses scenes at random (for statistical and testing purposes only).
NearestNeighborCartesianObjects	Implements the distance calculation algorithm using the continuous-form data (Section 6.2.1), first sorting objects by type. Object weights are supported (default equally weighted). Goals and players are further sorted by team. Identifying markers of flags and lines are not considered (i.e. a 30-meter flag and a goal corner flag are both simply considered flags). Implementation uses the greedy "skiers and skis" object matching algorithm (Section 6.2.4).
NearestNeighborCartesianCell-Objects	Same as above except implements the cell-based, discretized-region distance calculation (Section 6.2.2).
NearestNeighborCellBallGoal-Distance	A cell-based distance algorithm that considers the distance and angles between the ball and the opponent goal; no other objects or weights are considered. Created by a graduate student project group [31] (Section C).

7.3.2 Action Selection Algorithms

Table 7.2 lists the four action selection algorithms implemented in the `SceneSelection` class. The more complex algorithms generally call upon the simpler ones in a hierarchical manner (for example, all of the algorithms eventually drill down to use `ChooseFirstValid`).

The methods in SceneSelection are passed sets of scenes as parameters, and their return value is a `Scene` object. The `Action` corresponding to the returned `Scene` is the one action chosen as a "best" match; however, it is up to the agent to either transmit this same action unchanged, or use this result to drive other decision-making processes (as an example, the agent could further analyze the returned `Scene` object to try and deduce what specific target the action was aimed at).

Table 7.2: SceneSelection Algorithms.

ChooseFirstValid	Choose the first plausible action from the set (ignoring scenes with no actions).
ChooseRandom	Choose any action at random.
VoteWeighted	Performs a weighted majority vote (with user-specifiable weights). Does not consider the action parameters, only its type.
VoteWeightedWithRandom	A combination of ChooseRandom and VoteWeighted; given n, chooses a weighted majority vote except for $n\%$ of the time, pick an action at random.

7.4 Log Capture and Scene Generation

The RoboCup server by default provides its own logs of all actions that take place during the course of a game. (This is the mechanism that allows games to be reviewed.) However, the log files generated by the server represent only a global, "spectator" view of objects and players on the field. These logs provide locations of players on the field, but do not capture any of the data being sent to individual players.

In order to capture this data, agents need to be individually captured and logged. The result should be a log file which captures the information being sent from the server to that agent, and the commands that are sent back to the server. Essentially, the log file captures what is seen, heard, and acted upon, from the point of view of each agent. The total impact on the server and agents must be considered to ensure that the logging process does not affect the results.

The data in the resulting log files must be parsed and converted into the Scene objects for use either in the scene agent or for further processing. The scenes are then saved into a separate file.

The following sections describe the tools (*LogServer* and *LogToScenes*) used; Figure 7.4 depicts the process.

7.4.1 Log Generation

Options for data logging include packet sniffing (isolating one agent on the network) or using a network analyzer. The most straightforward approach is a log capture tool called *LogServer* [51], specifically designed to capture data traffic between the RoboCup clients and the server.

LogServer, a utility written by Paul Marlow, captures the data traffic between the

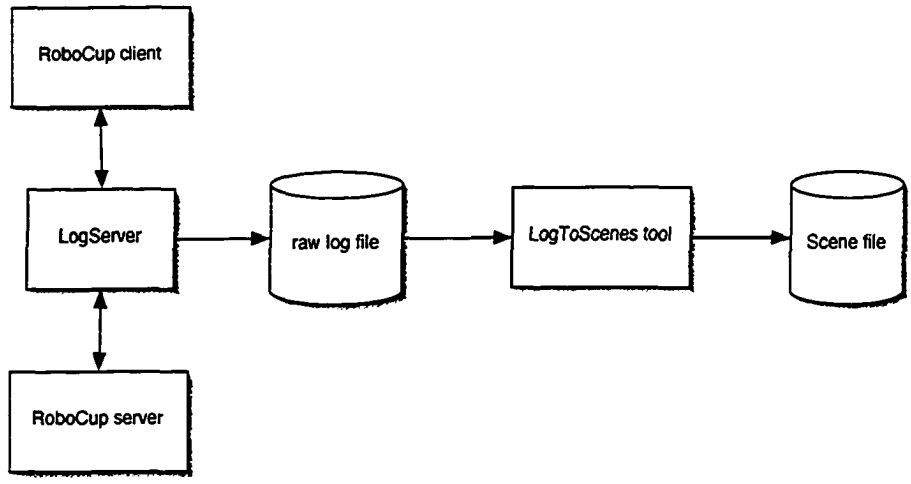


Figure 7.4: The data capture and scene generation process.

RoboCup server and all its clients by acting as a proxy between the clients and the server. Instead of connecting directly to the RoboCup server, clients are provided the IP and port number for the LogServer. LogServer accepts these incoming connections and forwards the data packets to the actual RoboCup server, while logging the data to a file. Responses from the server are likewise routed back to the originating client. One running instance of LogServer can log data from one, several, or even all of the RoboCup player clients at once. Multiple LogServers could also be run across multiple machines to provide better load distribution. The LogServer utility exists as a Java application as well as a Windows compiled executable. LogServer's impact on system performance is minimal [51]; the behaviour of the observed agent is not significantly impacted by the presence of the logging proxy.

The resulting output files from LogServer contain exactly (and only) the desired exchanges between the clients and server, which can then be parsed and converted to scenes.

7.4.2 Conversion to Scenes

The `LogToScenes` class is a utility program which takes as input a file containing logged data, and produces a file containing a set of `Scene` objects (actually a `Vector` of `Scenes`) as output.

LogToScenes analyzes each time cycle from the log (typically up to 6,000) and for each time cycle generates a `Scene` object containing both the corresponding `VisualInfo` object (listing all of the individual objects and their parameters; this class is reused from the Krislet code), and the corresponding `Action` objects containing the action that was taken. Actions are grouped with the RoboCup `see` message immediately preceding them. If multiple `see` messages exist for the same time period (indicating a pause in the server, such as before kick-offs or before the game is started), only the most recent one is kept.

The `LogToScenes` class contains a static `parseSeeInfo()` method which does the actual tagging of objects with the appropriate $(column, row)$ value according to the virtual table imposed by cell-discretization. This method is also called by the KrisletScenes agent to perform the same function in real-time with incoming server data, when the distance calculation algorithm requires it.

The `LogToScenes` utility supports generation of scenes using a user-specified cell-discretization size. It defaults to a size of $(5, 3)$ representing three discrete distances (adjacent, near, far) and five evenly-spaced angle "slices" (Section 5.4). Table 7.3 lists the default distance boundary value intervals.

The scene output file is generated using Java object serialization. (Disadvantages are large file size and lack of any process to manually edit the scenes once they are written. Also, serialized objects are tied to code versions — if the object code changes,

Table 7.3: Default distance intervals for scene discretization with LogToScenes.

Slice	Interval	Interpretation
0	$0 < x \leq 1$ meter	ball-kickable region
1	$1 < x \leq 5$ meters	close to player
2	$x > 5$ meters	far away from player

all saved files are deprecated.) Future versions may use more user-friendly files such as XML.

In addition to the object file, a human-readable ASCII output file is also created (Figure 7.5). Scenes are represented as tables of cells which show the counts of separate objects (players, lines, flags, ball, goal) in each cell. Actions and their parameters are also listed.

```
Scene 5984
Team Poland
# of Teammates/Oppos/Unknown = 2/5/2
  table:
+---------+---------+---------+---------+---------+
|         |       1L|BG 6P8F  |         |         |
+---------+---------+---------+---------+---------+
|   1P    |         |   1P    |         |         |
+---------+---------+---------+---------+---------+
|         |   1P    |         |         |         |
+---------+---------+---------+---------+---------+

  action:[dash]
  actionDirection: 0.0  actionPower: 110.0
```

Figure 7.5: ASCII representation of a scene.

All observed scenes are stored as potential match candidates. However, the LogToScenes utility automatically prunes non-useful scenes such as those that are

missing `VisualInfo` or `Action` objects, generally due to errors or noise in the log data.

7.5 Validation and Testing

For testing purposes, a `Validator` class was implemented using a structure very similar to the KrisletScenes' `Brain` class. It implements largely the same logic as the regular KrisletScenes agent (Section 7.2.4), but instead of connecting with a RoboCup server and processing incoming visual data in real-time, the `Validator` uses input from a second scene file which is loaded into memory.

The `Validator` execution algorithm is as follows:

```
parse command-line parameters
load the Knowledge scene file from disk
load the Validation scene file from disk
for every scene x in the Validation set:
     perform distance calculation using x as a key scene
     perform action selection
     get best action a
     compare a with action stored with scene x
     update statistics
end loop
display statistics
```

The `Validator` class provides a method for unit testing, statistical testing and cross-validation and is used extensively in Chapter 8. The algorithm allows the scene recognition algorithms to execute on the previously-captured, known scenes as input, and to compare the generated decisions with what had previously been associated with those scenes. If algorithms are operating correctly (both programmatically correct and logically correct), the generated results should be very close to the original

observations. Using the same scene file for both the validation input and the scene knowledge file should results in every scene file being matched against itself, and serves as a unit test to assert the scene recognition's correctness.

Statistics kept include the following:

- average execution time
- number of stored scenes
- total number of action comparisons
- number of "correct" generated actions (chosen actions match with the stored actions)
- number of correctly-matched *kick* actions (chosen due to the relative rarity, yet importance, of this action)

7.6 KrisletScenes Operation

The final KrisletScenes agent combines the modifications to the original Krislet, the new `Scene` and `Action` classes, the scene-based `Brain`, the collection of `DistanceCalculation` and `SceneSelection` algorithms and the `Validator` class. The `LogToScenes` class remains a separate standalone utility, though the two work together since the agent relies on the conversion method defined in `LogToScenes`.

All of the classes are combined into one Java package, called `visiontable`.

The client can be run either as an active RoboCup agent or in test/validation mode, with many options configurable from the command line. Table 7.4 describes the available parameters.

Table 7.4: Command-line parameters for configuring the KrisletScenes agent.

Parameter	Value	Description
host	*name*	server hostname (default localhost)
port	*num*	server port (default 6000)
team	*name*	team name (no spaces; default Poland)
scene	*filename*	file containing scenes to read (default krislet.scene)
sceneSel	*int*	ActionSelection algorithm to use (default 2): (0) ChooseFirstValid (1) ChooseRandom (2) VoteWeighted (3) VoteWeightedWithRandom
distCal	*int*	DistanceCalculation algorithm to use (default 1): (0) NearestNeighborCartesianObjects (1) NearestNeighborCartesianCellObjects (2) NearestNeighborCellBallGoalDistance (3) RandomDistance
numBest	k	value for k-nearest-neighbor selection
objectWeights	$\{n_1, n_2, \cdots, n_8\}$	weights for objects (default {1,0,0,0,0,0,0,0})[a]: {ball, goal, flag, line, allplayers, teammates, opponents, unknownplayers}
actionWeights	$\{n_1, n_2, \cdots, n_7\}$	weights for actions (default {1,1,1,1,1,1,1})[b]: {none, dash, kick, turnneck, turn, catch, move}
validate	*filename*	enters Validation mode using given scene file as test data

[a]Numbers separated by commas or slashes; braces optional. Do not use spaces!
[b]Same format as above.

7.7 Conclusions

This chapter summarizes the development of a Java-based framework for working with scenes, including the following:

- `LogToScenes`, a scene generation utility written in Java that reads captured log files and generates stored scene files;
- `KrisletScenes`, a scene-based RoboCup client, based on the Krislet agent [40], which uses distance calculation and action selection algorithms to perform basic scene recognition in real-time;
- a base set of `DistanceCalculation` and `SceneSelection` algorithms encompassing most major facets of scene recognition as explored in Chapter 6;
- a `Validator` module to perform unit testing and statistical performance analysis;
- an editing and visualization GUI tool for working with scenes;

Appendix C describes the development of a number of scene enhancements through student course projects at Carleton University.

The following chapter describes the results of tests run to evaluate the performance of the KrisletScenes client, and Chapter 9 presents suggestions for future work on the framework to improve future performance.

Chapter 8

Experimental Results

> Results! Why, man, I have gotten a lot of results. I know several thousand things that won't work.
>
> *Thomas Edison (1847-1931)*

Section 4.1 described the methodology we have used to develop, test and train our scene-recognition agent. This chapter revisits the last few steps of the methodology, describing a number of experiments developed to test and train the agent and to evaluate its effectiveness. We examine the results from these experiments and offer suggestions for improving the performance of the agent.

8.1 Experimental Process

The procedure for running tests on the imitative agent framework are:

1. *Agent observation and data capture.* Select a set of existing RoboCup soccer agents based on established criteria. Play some RoboCup soccer games and capture each player's I/O to a log file.

2. *Generate scenes.* Using the "scene" format (Section 5.2) and the implementation and conversion tools (Chapter 7), convert the raw captured data into scene files.

3. *Vary experimental parameters.* Load a scene file into the scene-based agent and connect the agent to the soccer server. Experiment methodologically with object and action weights, k-value, action selection and distance calculation algorithms.

   - *Qualitative performance analysis.* While experimenting with parameter values, observe the imitative agent on the soccer field. Evaluate the extent which its behaviour appears similar to that of the original agent.
   - *Quantitative performance analysis.* Using the Validator test tool, gather statistics measuring the imitative agent's performance using the original scene data as a training and testing set.

The following sections describe these steps in detail.

8.2 Agent Observation

8.2.1 Agent Selection Criteria

As previously discussed in Section 4.2, the ideal "observable" agents should be ones whose actions are primarily based on reactions to the "visual" inputs from the environment. Agents that rely heavily on memory, internal states, or inter-player communication, are difficult or impossible to imitate simply by observation of visual inputs and outputs.

Thus, in order to obtain the best results for imitation, the observed agents should meet the following criteria:

- Homogeneous player types. If a specific player type is observed, the imitative agent should be expected only to play in similar circumstances. The observed agent should not dynamically switch playing styles.
- Deterministic. The player should react the same way in the same (spatial) situations — the more predictably it does so, the stronger the result will be for the scene recognition algorithm.
- Primarily field-based. Primary factors driving the player's decision should be the positions of objects on the field, the only inputs available to the imitating agent. If the observed agent relies on many "invisible" factors (internal states, memory, communication), this will reduce the accuracy of the learning and recognition process.

To aid incremental testing and development, as well as to clearly see the level of effectiveness of the generated agent and its limitations, clients should be chosen with varying levels of complexity. This will help to expose the capabilities and deficiencies of the scene recognition algorithm.

8.2.2 Krislet, NewKrislet and CMUnited

Based on the functional requirements set out above, three different RoboCup client agents were chosen: Krislet [40], NewKrislet [32], and CMUnited [70, 69, 71].

The three teams were selected to represent the extremes in complexity and one intermediate. Krislet representing a team using simple decision-tree logic (see also

Section 3.1.1) and CMUnited representing a team with complex behaviours including formations and inter-agent communications. NewKrislet is an enhancement to the Krislet client which allows additional state-based behavior logic.

Krislet

Krislet's behaviour can be described using a set of rules, internally hard-coded in decision tree style logic (if/then statements):

1. If the ball is not within sight, turn 40 degrees.
2. If the ball is within sight but far away, and player is not facing the ball directly, face the ball.
3. If facing toward the ball but it is far away, run toward the ball.
4. If the ball is close by but the goal is not visible, turn 40 degrees.
5. If the ball is close by and the goal is visible, kick the ball toward the goal.

While unsophisticated, the Krislet algorithm is often effective even in competition against more complex agents. Krislet is stateless and deterministic as a result of following all of these decision rules at all times during the game.

NewKrislet (AntiSimpleton Attacker)

The NewKrislet client allows player actions to be specified using a push-down automaton model (still one of the "simple" clients as described in Section 3.1.1). Its package includes two sample versions of the NewKrislet Brain, dubbed the Simpleton and the AntiSimpleton. The Simpleton client is simply a reimplementation of the

Krislet algorithm using the PDA model, while the AntiSimpleton client implements a simple attacker/defender strategy aimed at attacking the weak points of the Krislet strategy [73].

The AntiSimpleton Attacker waits near the midfield line until the defenders pass the ball toward it, at which point it attempts to score a goal (much like Krislet). The Defenders wait in their designated home positions (forming a straight line from the goal toward the midfield) and watch for the ball to come within a reasonable distance. Once the ball is nearby, the Defenders will run toward it and kick it toward the Attacker[1].

All players, once their immediate goal is executed, return to their home positions and wait again for further activity.

The NewKrislet agent represents a client that is state-based and deterministic (the rules are explicit and always apply, with different sets of rules in place for different states). Each player type is composed of several states (*return to home*, *attack goal*, *clear ball*, etc. Because the Defenders spend a large proportion of their time at rest, the Attacker is chosen for observation.

CMUnited

CMUnited is an award-winning RoboCup soccer team (RoboCup-99 Simulator League World Champion) which uses a number of strategies including inter-agent communications, formation strategies, and a layered-learning architecture that provides the player with skills such as passing and dribbling. It includes features such as (from [69]):

[1]Our in-game observations note that the Defenders don't actually kick the ball toward the actual Attacker, but instead in the direction of where the Attacker's home position is. The Defenders assume this is where the Attacker will be.

- Hierarchical machine learning (Layered learning)
- Flexible, adaptive formations (Locker-room agreement)
- Single-channel, low-bandwidth communication
- Predictive, locally optimal skills (PLOS)
- Strategic positioning using attraction and repulsion (SPAR)

The base CMUnited source code forms the basis of many other RoboCup agents (e.g. [58]).

This team is chosen not because of any realistic hope of properly emulating the entire team but to determine whether any parts of its behaviour can be consistently emulated, thus considering the possibility that, when "imitation" is not possible, the imitative agent might still "learn" something of value from observation of a more complex client.

8.2.3 Data Capture and Scene Generation

The Krislet and NewKrislet teams were run on a 1 GHz Apple PowerBook G4 running Mac OS X 10.3.7 (Java version 1.4.2_05). The server (version 9.3.7 base code with 9.4.5 updates, compiled for OS X). The LogServer utility (Windows version) was executed on an AMD Athlon 750 MHz machine running Windows XP Professional; the two machines were networked via 10/100 Ethernet.

To prevent performance degradation (due to all the clients running independently as Java programs) the teams consisted of five players each. (The NewKrislet-AntiSimpleton team uses one Attacker and four Defenders per game). Several full

games were logged with the Krislet client and again with the NewKrislet-AntiSimpleton team. Games were typically run Krislet vs. Krislet, or NewKrislet vs. NewKrislet, though one game was also executed with Krislet vs. NewKrislet).

Finally, the `LogToScenes` utility was used to translate each log file into a stored-scene file (named `krislet1.scene`, `krislet2.scene`, etc.) A size of $(5, 3)$ rows and columns was used for object region discretization, corresponding with the examples from previous chapters.

The CMUnited team was executed and log-captured under similar circumstances by one of the student groups [76], using an AMD Athlon XP2000 machine running Linux. Rather than recapture and reconvert the CMUnited team (which has no OS X native binaries) the original captured scene files from the student project were reused[2].

8.3 Evaluation Method

The following sections describe in detail the testing procedures used to evaluate the performance of the imitative agent framework.

8.3.1 Statistical Results

We first define some statistical measurements which can be used to determine the effectiveness (prediction accuracy) of the scene agent. Given the large amount of observed data which can be used as a training and testing set, it makes sense to perform a type of cross-validation testing (whereby part of the training data is also

[2]A conversion pass was necessary because the `Scene` code had been updated, so a utility was used to load the student-generated scene files and re-save them, also using Java object serialization, into new scene files using the updated `Scene` class.

used as testing data). The `Validator` class makes this simple to do. For each test, several trials are run (typically five), each using a different portion of the data for training and then for testing. Results obtained are an average of each of the trials.

For validation and unit testing, the agent loads a collection of scenes into its memory and uses the Validator to test itself against the same scene file. Naturally, the distance algorithms should find a perfect match between what is currently seen and what was stored, and the action chosen by the agent should be the same as the action originally stored in the scene file. The Validator keeps a tally of hits/misses.

In addition to the tally of how many actions are correctly chosen by the agent, another tally of successful kicks is kept. Since intuitively a soccer-playing agent should know how to handle the ball and score goals, it can be argued that kicking is one of the most important actions the player can take — if the agent learns to run around the field but never properly kicks at the ball, it is not a very useful soccer playing agent. In instances where the original player kicked the ball, it may be helpful to determine whether the imitating agent also decided to kick.

For predictive testing, the agent loads one collection of scenes and is tested against a set of different scene files from the same team. This is a truer test of how well the distance algorithm can find a "similar" match, and determines whether, in any given situation, the player agent will make the same decision as the original observed agent.

8.3.2 Qualitative Observation

Statistical measures will not be sufficient to determine whether the agent is properly imitating the original player. While correct actions may be taken on a scene by scene basis, this may have little bearing on the actual in-game behaviour, where correct

actions must be taken in the correct order, among other factors. Ultimately the agent must perform actions that, taken as a whole, bear some level of similarity to the actions taken by the original observed agent.

To this end, a qualitative measure of the performance of the scene-based agent must be considered as a primary measure of success. Can similarities be detected between the actions of the imitative agent and the recorded action of the original? How closely matched is the general behaviour? Does the agent "play like" the original?

It is tempting to consider whether the imitative agent is playing a good game of soccer, but it must be noted that success here is determined not by how well the agent plays soccer on its own merit, but by how like the original the agent plays.

Depending on the complexity of the client and the success of the scene-based imitation, the agent may even be able to pass a form of "Turing Test"[3] if a bystander watching the output of the game was not able to determine which of two teams was composed of the original and which was the emulated agents.

Several tests were run, to measure the effects of:

- Varying the distance calculation algorithm
- Use of continuous versus discrete distance information
- Varying object weights to consider or ignore particular classes of objects
- Varying the scene selection algorithm and k value for k-nearest-neighbor selection

Other factors to consider include:

[3]Originally described by Alan Turing in his 1950 paper *Computing machinery and intelligence*; a machine is said to pass the test if a judge engaged in a natural language conversation with the machine and a human, and could not reliably tell which is which.

- Predictive measure statistics for both general actions and kicks
- Execution time (can a reasonable number of scenes be scanned in the 100 ms time interval between server "ticks")
- Quality of play of the imitating agent

8.4 Parameter Constraints

Between selection algorithms, object weights, action weights, and different table sizes for region discretization, for each of the three observed clients, there are many variables to consider. For this initial stage of statistical testing, the following limits are placed on data values to reduce the total number of test cases:

- Scene discretization is fixed at $(5, 3)$ columns and rows.
- Object weights are limited to $w \in \{0, 1\}$.
- Not all combinations of object weights are tested (2^8 or 256 total combinations); we consider each object class independently and then successively add object types.
- Action weights are set equally, $w = 1$.
- For scene selections, k is limited to $k \in \{1, 5, 15\}$.

The initial results of these tests should be sufficient to provide some analysis on what areas of scene representation and recognition to improve on in future development.

8.5 Results

The following sections describe and interpret the results obtained from experimentation. Data was captured into a Microsoft Excel spreadsheet as tests were run; the test results shown are averages of values taken over five test runs. Appendix A shows several examples of the original spreadsheet data before averaging.

8.5.1 Distance Calculation Algorithm

The following tests were run using each of the different distance calculation algorithms, with all the other parameters constant. In these tests, $k = 1$ and all object weights are set to 1 (players sorted into teammates/opponents/unknown).

Table 8.1, Table 8.2 and Table 8.3 show the results for Krislet, CMUnited, and NewKrislet-AntiSimpleton (henceforth referred to simply as NKAS).

Table 8.1: Effect of Distance Calculation variation for imitation of Krislet. $k = 1$, All Objects.

Krislet Distance Calculation	Training		Predictive		
	Action%	Kick%	Action%	Kick%	Max Size
RandomDistance	61.876	1.053	61.743	1.053	n/a[a]
NNCartesianObjects	99.987	100.000	64.414	12.149	2099
NNCartesianCellObjects	91.368	84.132	64.098	3.333	1611
NNCellBallGoal[b]	99.386	88.579	93.075	68.596	28865

[a]Independent of scene collection size.
[b]Object weights fixed, $w_{ball} = w_{goal} = 1$, all others 0.

The "RandomDistance" calculation is used to establish a floor by which to compare the other algorithms. As expected, the algorithms provide a very high predictive success rate in each case when the agent is tested on its own training data, though

Table 8.2: Effect of Distance Calculation variation for imitation of CMUnited. $k = 1$, All Objects.

CMUnited Distance Calculation	Training		Predictive		
	Action%	Kick%	Action%	Kick%	Max Size
RandomDistance	54.190	0.476	40.125	0.444	n/a[a]
NNCartesianObjects	99.954	100.000	44.436	35.159	563
NNCartesianCellObjects	94.447	99.079	43.760	11.571	825
NNCellBallGoal[b]	70.203	72.508	44.010	37.587	11503

[a]Independent of scene collection size.
[b]Object weights fixed, $w_{ball} = w_{goal} = 1$, all others 0.

Table 8.3: Effect of Distance Calculation variation for imitation of NKAS-Attacker. $k = 1$, All Objects.

NKAS-Attacker Distance Calculation	Training		Predictive		
	Action%	Kick%	Action%	Kick%	Max Size
RandomDistance	63.410	1.815	64.091	3.883	n/a[a]
NNCartesianObjects	100.000	100.000	68.746	8.040	1843
NNCartesianCellObjects	93.115	82.423	68.005	5.197	1116
NNCellBallGoal[b]	89.672	88.135	70.369	32.860	43903

[a]Independent of scene collection size.
[b]Object weights fixed, $w_{ball} = w_{goal} = 1$, all others 0.

this percentage drops considerably when used on actual previously-unseen test data. The success rate also predictably drops as the algorithms become more generalized (e.g. introducing the cell-based discretization).

The data takes on more meaning when graphed (Figure 8.1, Figure 8.2, and Figure 8.3), which shows a side-by-side comparison between the different calculation algorithms in each of the training and testing situations.

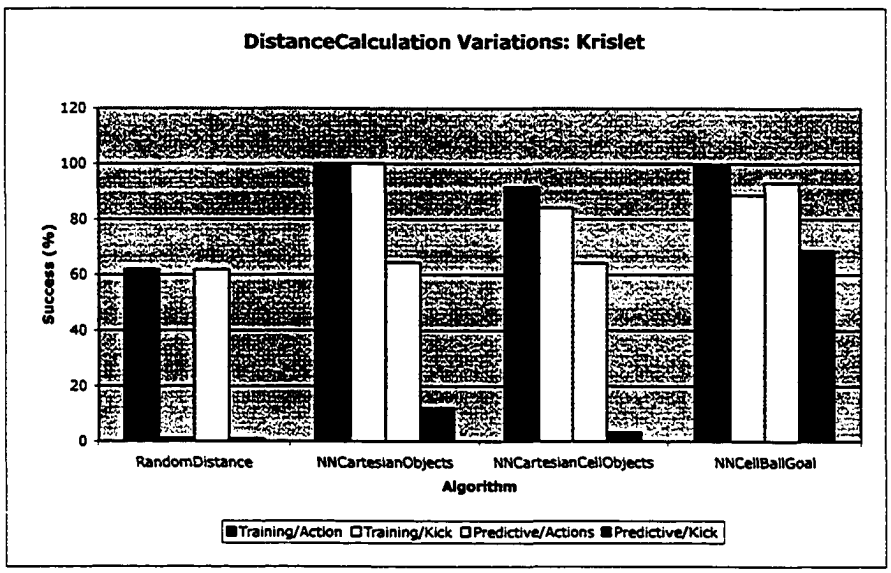


Figure 8.1: Effects of Distance Calculation algorithm variation for Krislet imitation.

It appears that predictive success drops with successive variations to the algorithm (though the NearestNeighborCellBallGoal algorithm seems to have a high predictive accuracy for Krislet, and is also more consistent between training and prediction for NKAS and CMUnited). These results appear to show that the introduction of data discretization worsens the performance, but the statistical results alone may not tell the whole story. A high success with training that drops significantly in the predictive

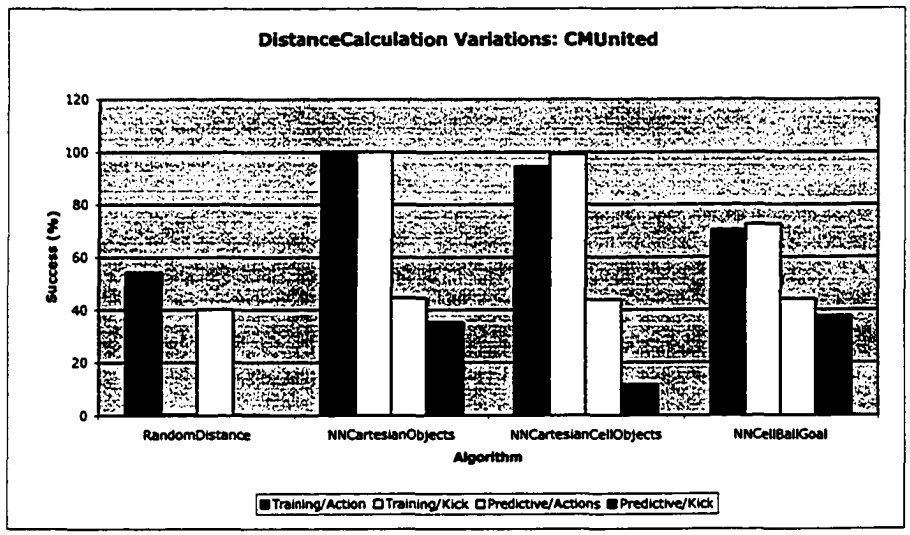


Figure 8.2: Effects of Distance Calculation algorithm variation for CMUnited imitation.

testing suggests that the continuous algorithm is overfitting the data. (Considering the numbers of objects to be considered in the scenes, such as lines and flags, this is not hard to imagine.)

Let us consider the qualitative behaviour of the agent itself — how well it models the agent it is imitating. Table 8.4 describes the observed results when the client is run with each of these algorithms. Tests were typically run with one agent on the field. In situations where the agent did not immediately begin playing, the ball was manually dropped in various positions to determine whether the agent would react to the position of the ball.

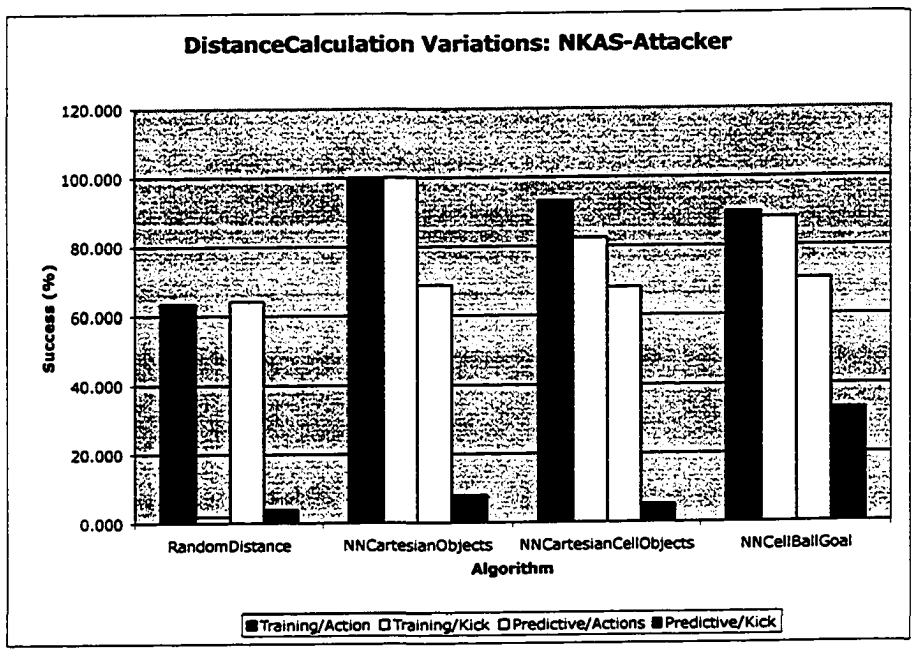


Figure 8.3: Effects of Distance Calculation algorithm variation for NKAS-Attacker imitation.

Table 8.4: Qualitative evaluation of the agent's imitative behaviour of the three recorded agents.

	Krislet	**CMUnited**	**NKAS-Attacker**
NNCartesian-Objects (continuous-path distance)	Appears to run in fits and starts (frequently gets stuck in a cycle of turns). Runs slowly in the general direction of the ball, but begins to wander once close to the ball.	Player moves somewhat and gets stuck in a loop standing on the field turning around in circles. No appreciable similarity.	Player slowly approaches a spot approximately mid-field, near the center circle, and once there begins to kick continuously. Independent of ball position, occasionally heads toward a new spot and repeats. Appears to be copying exact positions where original agent was when it kicked the ball.
NNCartesianCell-Objects (discretized cell distance)	Appears to run in fits and starts. Pauses (as it sits and turns) even more frequently. Does not seem to notice the position of the ball. No appreciable similarity.	Player moves and gets stuck in a loop turning in circles while stationary on the field. Occasionally leaves its position to go elsewhere before resuming turning on the spot. Shows no inclination to follow the ball.	Player makes a beeline toward the goal; once it approaches the goal, kicks, turns around, runs toward its own side. Once past midfield, turns around, heads toward goal again, and repeats. Appears to be copying the kick-then-go-home behaviour of the original agent (but ignores the ball position).

Continued on next page ...

Table 8.4 – continued from previous page

	Krislet	**CMUnited**	**NKAS-Attacker**
NNCellBallGoal (object-specialized variation of a cell-based distance, considering ball and goal only)	Runs toward the ball, turning occasionally to reorient itself. Once at the ball, usually (but not always) turns toward the opponent goal and attempts to kick. Occasionally tries to kick when actually outside the kickable radius, and gets stuck in an infinite loop until the ball is moved – then it turns and chases the ball again. **Strong resemblance to original!**	Player exhibits some stationary spinning but also runs appreciable distances in between spin "cycles". Responds to movement of the ball by facing it and running toward it; sometimes it will reach the ball and attempt to kick it, other times it will begin wandering elsewhere.	Runs toward the ball fairly steadily, kicks at it. Sometimes "stuck" just in front of the ball (turning left/right, or kicking); sometimes successfully kicks. Aim is not perfect, sometimes in completely wrong direction. Once kicked, begins to head away before returning and repeating.

Observations

The players do not appear to react very well using the general-purpose distance calculation algorithms; the behaviour using these was not much improved over simple random action selection (indeed, for Krislet, choosing "random" behaviour seemed more natural than using either of the two CartesianObject distances). This can be attributed both to oversimplification of the object pairing algorithm and the overfitting of data. Performance was much more significant — and the agents response snappier — using the NearestNeighborCellBallGoal algorithm, which produced very good results, especially with Krislet. (This should be expected, since the algorithm considers only the two objects known to be used by Krislet for its simple decision-making engine). For NKAS, behaviour also showed a strong resemblance to the original, though unlike the original, the agent did not stop and wait once it reached a home position, but instead would run back and forth constantly.

Timing is another consideration — although the scene database contains typically 2000-3000 scenes, the time spent to scan all of them for all object types becomes prohibitive with some of the distance calculation algorithms (though performance is naturally tied to the CPU; in this case, the Java implemention on Mac OS X is known to be somewhat slow.) The data tables show an estimated number of total scenes that could be scanned in the 100 ms (typical) that an agent should respond within; the optimized distance calculation is dramatically faster than the other algorithms (except random selection).

This limited success (and the much higher success of the specialized NearestNeighborCellBallGoal algorithm) points toward the next area of optimization, object weights. Clearly if data is being overfitted, then more generalization needs to take

place — perhaps by removing certain classes of objects that do not need to be considered. The following section considers these possibilities.

8.5.2 Object Weights

Object weights were tested where $w \in \{0, 1\}$, considering only the presence or absence of particular classes of objects.Objects were tested by considering the predictive power of each class of objects on its own and in combination with others (to see whether the presence or absence of a class of objects impacts the accuracy of the prediction).

Table 8.5, Table 8.6 and Table 8.7 show the results of these tests for each of Krislet, CMUnited and NKAS.

Table 8.5: Effect of object weights on the matching algorithm for imitation of Krislet. $k = 1$, NearestNeighborCartesianObjects distance.

Krislet Test Description	Training		Predictive		
	Action%	Kick%	Action%	Kick%	Max Size
Random Distribution	61.876	1.053	61.743	1.053	n/a[a]
Goal Only	86.668	75.658	63.809	4.167	-[b]
Flags Only	98.743	94.737	66.101	5.921	-
Lines Only	91.726	66.447	59.246	0.000	-
Players Only	98.638	100.000	65.337	11.842	-
Ball	99.407	90.684	97.992	35.000	13998
Ball/Goal	99.941	100.000	75.957	24.342	8033
Ball/Goal/Flag	99.980	100.000	66.395	5.263	2785
Ball/Goal/Flag/Line	99.980	100.000	65.700	5.263	2251
Ball/Goal/Flag/Line/AllP	99.987	100.000	66.210	6.053	1661
All Objects	99.987	100.000	64.414	12.149	2099

[a]Independent of scene collection size.
[b]Not measured.

As before, the test sets show a near-100% match rate with the training data sets,

Table 8.6: Effect of object weights on the matching algorithm for imitation of CMUnited. $k = 1$, NearestNeighborCartesianObjects distance.

CMUnited Test Description	Training		Predictive		
	Action%	Kick%	Action%	Kick%	Max Size
Random Distribution	54.190	0.476	40.125	0.444	n/a[a]
Goal Only	65.127	18.254	49.961	0.000	-[b]
Flags Only	99.147	100.000	56.884	13.228	-
Lines Only	85.933	82.143	53.264	2.275	-
Players Only	94.373	89.286	53.825	7.672	-
Ball	68.262	78.873	41.904	58.683	5887
Ball/Goal	83.281	86.730	43.738	35.333	8554
Ball/Goal/Flag	99.793	100.000	44.865	12.968	2293
Ball/Goal/Flag/Line	99.867	100.000	44.540	33.381	1904
Ball/Goal/Flag/Line/AllP	99.946	100.000	44.062	9.492	1426
All Objects	99.954	100.000	44.436	35.159	563

[a]Independent of scene collection size.
[b]Not measured.

though this value drops when some object classes are considered, suggesting that the positions of these objects do not account for much in the original agent's decision-making. Typically the success rates are not much more than simply picking actions at random. As more object classes are considered, data overfitting begins to occur, hence the many 100.000% success rates.

Again, the data is best shown visually, as Figure 8.4, Figure 8.5 and Figure 8.6 show. Only the Predictive data sets are graphed from the tables; the Training results are not plotted since many of them are very similar, and on average are close to 100% anyway.

In the Krislet imitation, the graph shows a similar level of predictive accuracy in all object configurations, but with a significant spike in predictive accuracy when considering just the ball, and a smaller spike when considering the ball and goal. This

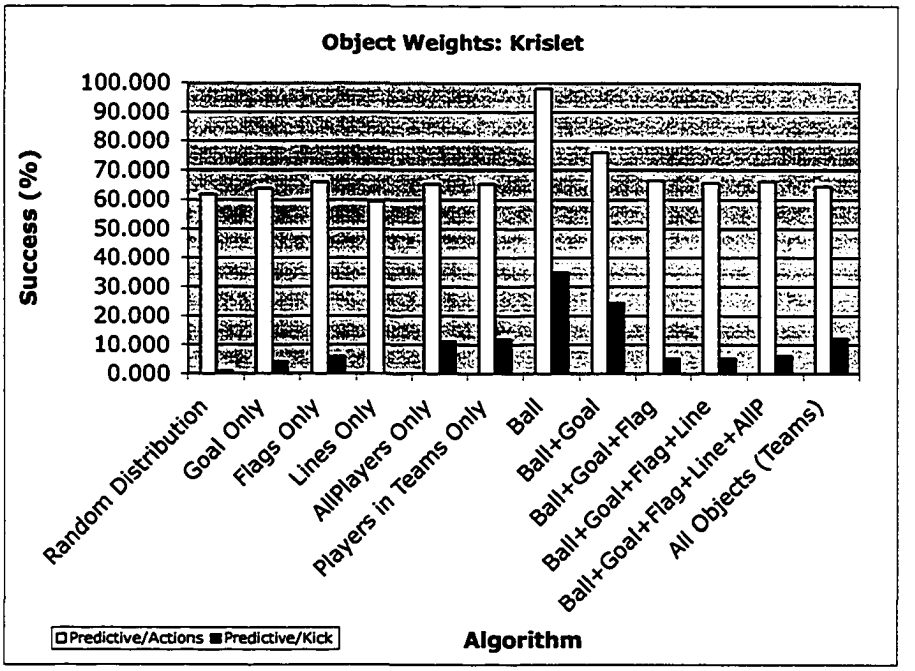


Figure 8.4: Effect of varying object weights for Krislet imitation.

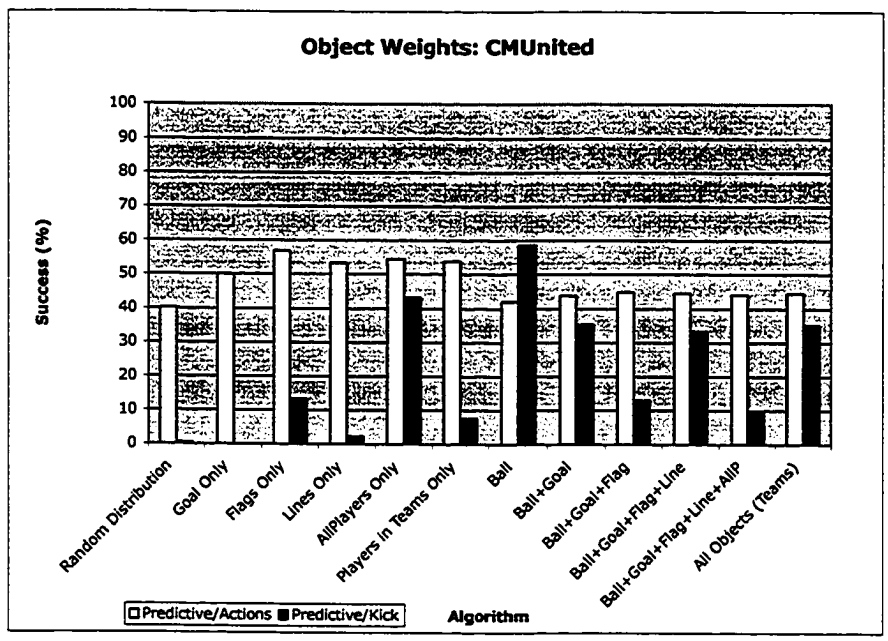


Figure 8.5: Effect of varying object weights for CMUnited imitation.

Table 8.7: Effect of object weights on the matching algorithm for imitation of NKAS-Attacker. $k = 1$, NearestNeighborCartesianObjects distance.

NKAS-Attacker Test Description	Training		Predictive		Max Size
	Action%	Kick%	Action%	Kick%	
Random Distribution	63.410	1.815	64.091	3.883	n/a[a]
Goal Only	86.326	40.360	70.697	0.966	-[b]
Flags Only	99.870	100.000	73.371	5.897	-
Lines Only	96.524	92.319	70.305	5.959	-
Players Only	89.319	44.739	70.806	0.966	-
Ball	75.986	87.922	63.333	32.590	22882
Ball/Goal	94.949	92.308	75.634	23.530	11557
Ball/Goal/Flag	100.260	100.000	70.785	6.183	2848
Ball/Goal/Flag/Line	99.978	100.000	70.658	6.183	2526
Ball/Goal/Flag/Line/AllP	100.000	100.000	68.142	6.880	2185
All Objects	100.000	100.000	68.746	8.040	1843

[a]Independent of scene collection size.
[b]Not measured.

is consistent with the hard-coded behaviour of Krislet — it does indeed only consider the ball and goal — but it is useful to note that this is clearly seen simply from the statistical data. All other object combinations are not significantly improved from simple random actions. Clearly, knowing the position of the ball is enough to predict the behavior of the Krislet agent with fairly high accuracy.

The analysis of the CMUnited data is not as clear — no one object grouping or set of objects can predict the agent's behaviour significantly better than random chance. The highest success rate seems to come from considering the positions of players or flags — this may be consistent with the team's known reliance on formations and passing. When considering just the "kick" action, considering the ball gives the best indication of when the player will kick, which is a natural conclusion.

Similarly, the NKAS agent data does not present any significant trends, though

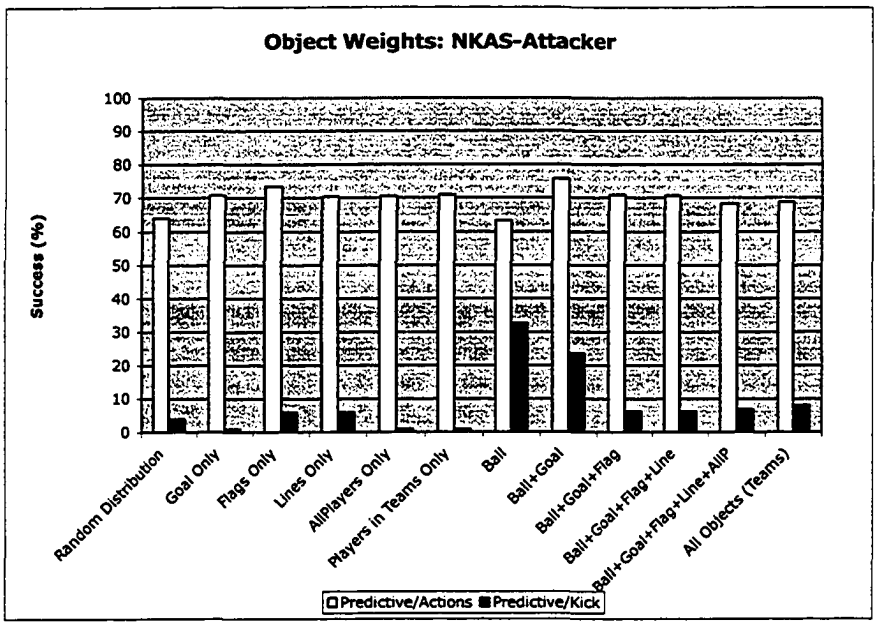


Figure 8.6: Effect of varying object weights for NKAS-Attacker imitation.

there are smaller spikes in accuracy when considering the ball, or the ball and the goal. Analyzing the flag objects appears to be only slightly less accurate, though the values are so close to each other that is difficult to determine whether this is a trend or simply random. It is worth noting that NKAS does use flags to determine its position on the field (as does CMUnited).

Next, we consider the effects of each object type on the qualitative behaviour of the agent. We combine these results with those in the previous section by considering both the discrete (cell-based) and continuous-information distance calculation algorithms. (We do not consider the NearestNeighborCellBallGoal, despite its previous success, since it is not written to support multiple object types and weights.) Tests were conducted by instantiating one agent alone in the field, except for the tests involving multiple players, in which four instances of the agent were run, two per team.

With NKAS, the original Defenders are instantiated to play along with the imitated Attacker.

Table 8.8 shows the results from this test.

Table 8.8: Qualitative results for object weight tests using scene files from the three observed agents.

	Krislet	**CMUnited**	**NKAS-Attacker**
Ball Only	Continuous: Player runs toward the ball and kicks it in a random direction, then repeats. Follows the ball wherever it is dropped. Looks much like Krislet except ball is not being aimed at goal. Discrete: Same as continuous, but player occasionally gets stuck just short of the ball and attempts to kick at in unsuccessfully.	In both continuous and discrete cases the agent is caught in a loop turning in circles! It does not appear to be driven by the position of the ball. No useful imitation here.	Continuous: Player reaches ball, kicks it away; by chance, was in position to receive a pass from teammate, sent it toward opponent goal. Generally looks like the original except kicking is not always aimed toward the goal. If agent is too far away from the ball, it only sits and "watches". Discrete: Player simply sits and turns.
Ball, Goal	Continuous: Player runs toward the ball, though slower and with much wavering (turning). Once it reaches the approximate location of the ball it turns toward the goal and attempts to kick. Sometimes this is successful; sometimes the agent is actually standing too far from the ball to kick successfully. "Reaction time" seems slow. Discrete: similar, though slower to act (characterized by more turns) Similarity is quite apparent otherwise.	Players spend a lot of time spinning in circles, occasionally wandering short distances. The discrete (cell) based player consistently wanders toward the ball, until it is right on it; it does not kick the ball. The player operating with the continuous-distance algorithm seems to wander the field, only vaguely impacted by the ball position.	Continuous: If too far away, agent wanders back and forth. When near the ball, agent runs to the ball, kicks it, continues to chase after it. Player's aim is not consistently toward the goal. Instead of chasing the ball, player sometimes turns away, then returns. When near goal, player tries to kick, but sometimes gets "stuck" doing so. Discrete: Player only runs to goal, then to ball, then to goal, etc.

Continued on next page ...

Table 8.8 – continued from previous page

	Krislet	CMUnited	NKAS-Attacker
Ball, Goal, Flags	Significantly slower search time. Agent wanders semi-randomly, sometimes seeking the ball, sometimes wandering a short radius from it. Discrete algorithm causes agent to stop and kick at nothing.	No useful similarity in either algorithm; the players wander aimlessly turning and turning their necks.	Player wanders the field back and forth, attempts to kick when near the ball (but is not near enough, and kick fails). Same, but slower, in discrete mode.
Ball, Goal, Flags, Lines	As above, agent wanders the field seemingly without aim, occasionally making a run toward the ball but overshooting and circling. Discrete algorithm stops and kicks.	As above, no useful similarity; players wander about, though more so than above. Player reacts slowly (i.e. algorithm is running slowly)	Player wanders, turns, randomly kicks, but does not visibly react to the presence of the ball. Slow reaction time.
Ball, Goal, Flags, Lines, Players	Agents wander aimlessly, vaguely showing awareness of ball location.	No useful similarity; players appear to wander the field or sit and turn as above.	Player wanders back and forth between the center and the goals, kicks randomly. Discrete mode much slower.
Ball, Goal, Flags, Lines, Players in Teams	Reaction is slow due to increased search time (smaller scene file or faster CPU would help this). As before, agents wander the field; the discrete modeling agent more so than the continuous one, which appears to favour the balls vicinity.	No useful similarity; players appear to wander the field or sit and turn as above.	Player wanders, randomly kicks, appears to be affected by other players but not in a discernible way (e.g. triggers new random direction of travel).

Observations

It is evident from these results that, as the statistical results suggest, the agents react to the presence of different types of objects in different ways. Clearly, not every type of object is relevant. In fact, trying to make a decision based too much on, say, the exact positions of lines and flags around the player, will result only in matches with scenes in which the original agent also stood in the same position on the field. This lack of generalization and the corresponding implication — a requirement to have a recorded scene from every agent in every conceivable field position — is clearly not practical. A better matching algorithm would need to consider some compromise between being too general ("a flag") and being too specific ("the top right flag at the 50-yard-line") when pairing objects to perform the distance calculation.

It can be seen that some objects — such as the ball and goal — have obvious importance to all three of the observed teams, both because they are distinct (there can only be one ball, and no more than two goals) and because of their known relevance to the objectives of soccer-playing agents. Though this is known from knowledge of the domain, this was also determined largely experimentally through both statistical and qualitative performance. When considering the ball and goal as equally weighted objects with all other objects weighted 0, the tests using Krislet and NKAS data show very similar performance to the NearestNeighborCellBallGoal algorithm which was shown earlier as being effective at imitating these two agents. Though no consistent similarity was found for CMUnited, some low-level skill aspects (such as running toward the ball) were noted.

It is also apparent that discretizing scenes and considering matches only based on cells seems to lower the predictive power as well as the agents performance. Discrete

algorithms tended to cause more instances of agents "getting stuck" — whether caught in a turn loop, or a kick loop, and in some cases it is evident that the point at which it gets stuck may correspond to cell boundaries, such as in the case of the player which tries to kick at the ball when it is not quite close enough to successfully do so. The notable exception is with CMUnited and some instances with NKAS, where the player's behaviour seemed more natural when using the discrete distance calculation.

So far it has been shown that, depending on the complexity of the original client, it is possible to come close in imitating its behaviour — if not close enough to play indistinguishably from the original, at least close enough to see facets of the way the original player "thinks", e.g. what types of objects it considers important when making its decisions. One final variable to consider is the variation of the k-nearest-neighbor algorithms and the scene selection algorithms used to choose the final actions to take.

8.5.3 Scene Selection Algorithm

We consider the effect of modifying the k-value and the specific algorithms used in the selection of the final candidate action from the matched list of k-nearest scenes. Scenes may be chosen from the list (set at $k \in \{1, 5, 15\}$) either randomly or by an equally weighted majority vote. (Note that when $k = 1$, the action selection algorithm doesn't matter, and the "Choose First Valid" strategy has the same effect regardless of the value of k.)

Table 8.9, Table 8.10 and Table 8.11 show the results for each of the selection strategies.

Predictably, the accuracy on the training data is reduced as k increases, and overall

Table 8.9: Effect of varying k-value and selection strategy for imitation of Krislet (using NearestNeighborCellBallGoal distance).

Krislet Test Description	Training		Predictive	
	Action%	Kick%	Action%	Kick%
$k = 1$	99.386	88.579	93.075	68.596
$k = 5$, equal weighted vote	94.018	57.368	89.088	41.447
$k = 5$, random vote	92.797	66.588	87.728	46.184
$k = 15$, equal weighted vote	85.188	60.298	82.795	51.316
$k = 15$, random vote	84.347	31.053	81.483	47.982

Table 8.10: Effect of varying k-value and selection strategy for imitation of CMUnited (using NearestNeighborCellBallGoal distance).

CMUnited Test Description	Training		Predictive	
	Action%	Kick%	Action%	Kick%
$k = 1$	70.203	72.508	44.010	37.587
$k = 5$, equal weighted vote	69.914	39.206	48.438	27.825
$k = 5$, random vote	63.518	36.873	45.106	33.206
$k = 15$, equal weighted vote	68.523	4.889	45.955	5.000
$k = 15$, random vote	61.942	11.714	44.885	12.286

Table 8.11: Effect of varying k-value and selection strategy for imitation of NKAS-Attacker (using NearestNeighborCellBallGoal distance).

NKAS-Attacker Test Description	Training		Predictive	
	Action%	Kick%	Action%	Kick%
$k = 1$	89.672	88.135	70.369	32.860
$k = 5$, equal weighted vote	88.485	40.808	73.387	22.003
$k = 5$, random vote	83.671	33.217	70.399	14.459
$k = 15$, equal weighted vote	85.629	9.408	83.338	3.972
$k = 15$, random vote	78.760	11.558	71.169	7.709

performance appears to decrease as k becomes large — since this has the effect of matching scenes that have less and less similarity to the scene being matched, this is expected. Note that while the predictive accuracy seems best when $k = 1$ for Krislet, that the other algorithms perform better (e.g. $k = 5$) for CMUnited and NKAS (though for NKAS, $k = 1$ helps it to kick more accurately). In all cases, "opening it up for vote" appears to reduce the chances that the agent will perform a kick action when it is supposed to. Since kicking the ball is generally performed less frequently than turning and dashing, this is expected, and suggests that different action weightings need to be considered if using a vote selection.

These results are reflected in qualitative analysis. To compare, two agents were instantiated, using (from the results obtained above) the discrete- and continuous-distance algorithms, considering only the ball and goal objects. For one agent, $k = 1$ (the control) and the other agent uses one of the other algorithms. The results are summarized in Table 8.12 below.

Table 8.12: Qualitative effects of Scene Selection and k-value on imitation of the three observed agents.

	Krislet	**CMUnited**	**NKAS-Attacker**
k=5, random vote	Although occasionally venturing in a random direction, the agent plays consistently better than the opponent, outscoring it and not getting caught in kick or turn loops for very long. At the ball, it does take a while before it performs the "correct" action but it eventually will. Opponent occasionally stops and "waits".	Using discrete distance, the agent does not appear to even consider the objects as it simply wanders off the field. Performance appears considerably worsened compared to when $k = 1$. Both continuous and discrete algorithms seem to generate the same results.	Player bears a good similarity to the original. Agent goes to ball, kicks it, turns and runs away; then returns and repeats. Capable of scoring goals consistently, though occasionally kicks too early, and aim is not always good.
k=15, random vote	As above, but wanders around more, gets caught in turn loops for longer, but still recovers. Slower to react and kick the ball when it reaches it.	The player wanders randomly, going off in a direction, turning, spinning in circles. When the ball is placed close by, sometimes it will run up to the ball, and if it doesnt wander away, it kicks the ball.	Similar to above, but the player wanders slightly more erratically and slower. Sometimes overshoots when approaching the ball.

Continued on next page ...

Table 8.12 – continued from previous page

	Krislet	**CMUnited**	**NKAS-Attacker**
k=5, equal weighted vote	Approaches the ball in fits and starts, but eventually tries to kick, though gets caught in cycle if not quite close enough. Not too distinguishable from opponent (more so in discrete mode than in continuous)	Wanders and turns. Does not seem affected by the location of the ball, though sometimes makes a dash in its general direction.	Player comes up to the ball, sometimes kicks, sometimes just turns around and returns. Reaches end of field before turning around and repeating the approach to the ball. Consistently scores goals, aim appears better than previous attempts. Slightly slower.
k=15, equal weighted vote	Player reaches the ball after a direct approach, gets closer and closer before either kicking it or overshooting and turning around. Opponent ($k = 1$) stuck in a loop.	While frequently stopping to turn, the player sometimes slowly works its way toward the location of the ball, but does not move further or kick.	Player runs back and forth more frequently, doesn't seem to reach the ball close enough to dribble or kick before the player turns around and runs back to its "corner" position. Didn't kick the ball at all.

Observations

It appears that the agents that incorporate an element of randomness succeed more often than those that do not, at least in Krislet and NKAS, where it turns the imitative quality from marginal to quite reasonable. Although statistically, choosing at random provides a lower success rate on individual scenes (which is also reflected in CMUnited), given enough time the randomizer eventually may pick the "correct" action. Since we consider behaviour over a sequence of time, it is eventual that the random algorithm will choose an action that appears correct (though it is of course just as likely to choose one that is incorrect).

This result demonstrates a potential flaw as well as its workaround. All other selection algorithms, other than the random scene selection, are completely deterministic. If the action chosen is one that does not significantly change the environment, the same scene and actions will continue to be chosen. This is why so many test runs fail when the imitative agent gets caught in a loop while it tries to turn, or kick. With a random selection, even the agent that gets caught forever trying to kick at a ball that is just beyond its reach will eventually, at random, head closer to the ball, thus allowing it to complete its intended action.

One solution is to allow an element of randomness into the selection process, or equip the agent with the ability to determine when the currently-selected action is clearly not working, so it can try something else. Another solution would be to design the imitative agent with a low-level skill base so that rather than simply (and blindly) imitating individual actions, the agent would have enough built-in logic to realize that, for example, a kick would fail since the ball is too far away. Note however that imparting extra domain knowledge to the client strays from the original objective

of having a hands-free and automated process.

The fact that a vote algorithm degraded the performance of the CMUnited test suggests that it may not be completely deterministic based on the object positions — in different situations, given the same set of objects around in (at least given the ball and goal) the CMUnited player chose completely different actions. Thus, a vote is insufficient and actually reduces the chance of picking the correct action.

8.5.4 Combined Results

The previous tests provide an opportunity to experiment with various combinations of parameters in a refinement of the "best" sets to use to imitate the observed agents. Although the eventual success varied, the trends appeared to be generally consistent. The best overall results are found when using the following combinations of parameters:

- Continuous, rather than discrete, distance calculation algorithm (such as NearestNeighborCartesianObjects), and appropriate variation in object weights, *or*
- A distance calculation algorithm more finely tuned to the specific objects expected to be used by the original agent (for example the NearestNeighborCellBallGoalDistance algorithm)
- $k = 5$ with random action selection (though an equal-weighted vote worked for NKAS), or $k = 1$ (which is equivalent to the "Choose First Valid" algorithm). Any other scene selection method would require a relatively small value of $k (k \leq 5)$ and experimental variation in action weights.

This combination of approaches produces an imitative agent that is capable of completely imitating the decision algorithms of the Krislet client and many aspects of the NewKrislet-AntiSimpleton Attacker with reasonable success. During one demonstration, the Krislet client was placed in competition with its imitation; from a spectator's perspective, the two agents were indistinguishable. Only the final score of the game revealed which one was the imitation, since it operated slightly slower than the actual Krislet agent and was thus repeatedly and heavily outscored.

In the case of the NewKrislet agent (and the particular implementation of the AntiSimpleton), much of the client's behaviour was observed correctly — such as running to the ball, kicking it toward the goal, and running back toward the home position. Where the algorithms fail is in determining when each action should be taken, which is a function not only of the object positions but also of the agent's internal state (although this might be alleviated somewhat by designing a Distance-Calculation algorithm that could consider or predict state variables).

The algorithms also fail when attempting to learn to behave like CMUnited. This was expected, since CMUnited contains logic which considers far more than simply the positions of objects on the field. Because of its internal layering and complexity, its behaviour cannot be predicted simply by considering the inputs and outputs to the agent (and once again, it is possible that performance may improve with a more complex imitative agent).

The results, and particularly the disparity between the statistical "successes" (high percentage values in the action and kick columns of the tables) and the actual result of the agent in a qualitative evaluation, suggest that it is not generally sufficient for an agent to simply blindly follow directions "pulled out of a hat" — almost literally,

in this case, where the actions simply come from the previously-observed collection of scenes. In simple cases, such as Krislet, this may work well, but even with an agent slightly more complex (the NewKrislet agent, with its addition of state variables) the difficulties become evident.

8.6 Conclusions

The observations presented in this chapter suggest that it is possible to learn to imitate the behaviour of a RoboCup client if its behaviour can be captured in a simple logical construct such as a decision tree or flowchart. Even so, it requires some experimentation and statistical testing before an effective set of algorithm parameters are chosen. The disparity between the statistical prediction results and the qualitative "play quality" results suggest that effective play is more than simply guessing correctly on a scene-by-scene basis. A more sophisticated evaluation formula, short of human intuition, is required in order to effectively calculate quantitatively how well the imitative agent performs as a whole.

Note that if such a formula or set of qualitative metrics was available, all of the experimentation, weight derivation, etc. could be entirely automated, resulting in an agent that could capture and analyze and "learn" — completely unattended by human domain experts. Currently, however, all that is required is for a discriminating observer to experiment with the various parameters until the best desired behaviour is evident.

Additionally, the imitative agent should have a base level of domain knowledge, at least enough to determine logically whether the chosen action (derived from the scene-recognition system) is likely to work or not. For example, if the selected action

is to kick, the agent should know enough about the physics of the RoboCup world to realize that if there is no ball within a kickable distance, there is no point in kicking. The agent should also be able to prune out actions that don't make sense when taken out of their context. For example, consider the CMUnited client. Its logs (and scene files) demonstrate that *turn_neck* occurs very frequently. One can assume that the original agent issues this command to receive visual information from a different angle, in order to aid its own internal world view. This is not an action which should be imitated indiscriminantly; indeed, it would probably damage the predictive ability of the imitating agent, since turning the players neck alters the visual information being received, and affecting the distance calculation process.

The current stateless, single-layered imitative agent is able to almost perfectly copy the behaviour of a similarly stateless, single-layered agent (Krislet) and is fairly representative of at least some of the behaviours of more complex clients. These results are generally encouraging and suggest that with further development (including a layer of base logic and basic skills, the ability to internally represent different agent states, etc.) it may be possible to further increase the accuracy of the imitative agent.

Chapter 9

Conclusions and Future Work

> If a man will begin with certainties, he will end in doubts; but if he will be content to begin with doubts, he will end in certainties.
>
> *Francis Bacon (1561-1626)*
> ADVANCEMENT OF LEARNING

The goal of this thesis is to address the problem of requiring large amounts of time and human effort to implement an agent, and thus posed the question: *Can a software agent learn, by observing and imitating the behaviour of another, in an automated process with little or no human intervention?*

A partial answer to this question was demonstrated with the development of the Scene representation format for spatial knowledge, several scene recognition and matching algorithms, and the simple KrisletScenes RoboCup agent designed to work with the scene knowledge. The agent is capable of imitating simple agents after observation and some simple parameter tweaking. Although the developed programs were able to imitate a few simple agent behaviours, they point the way to many potential improvements which would increase the functionality of the learning agent and improve its success in learning from observation.

9.1 Contributions

The following contributions were made to the RoboCup, AI, and ML research communities:

- *Scenes* — A spatial knowledge representation (derivative from the Krislet agent) with both continuous-space and discretized-space properties
- *A software framework* to support scene-matching and selection using a k-nearest-neighbor approach and customizable algorithm implementations
- *A set of algorithms* implementing basic k-nearest-neighbor searches using the Scene format, performing object-matching with a simple heuristic approach (based on the skiers-and-skis problem)
- *A RoboCup client*, KrisletScenes, which is based on the scene knowledge framework and the scene recognition algorithms
- *A set of tools* to evaluate and validate scene algorithms and to generate scenes from log files.

We have developed and implemented a framework which provides the ability to experiment, from a broad perspective, on the concept of scene generation and recognition, in a process that could potentially be fully automated. The initial tests show viable results, in the form of an imitative agent that is capable of reproducing, with fairly good results, the logic of the Krislet client, and slightly less successful (but still clearly derivative) of the NewKrislet client. With more sophisticated distance calculations and/or object matching algorithms (see Section 9.3 for discussion of potential future work), the success rate should continue to improve.

9.2 Limitations

There are a number of limitations in the current agent framework that must be addressed:

- *Limited inputs.* The current implementation is limited only to current visual information, and does not consider potential other stimuli such as body state (stamina), game states, or any memory (including previous visual states). (It also does not consider messages from a coach or other players, though this is intractable anyway, without foreknowledge of the communications protocol). The agent can only be as flexible as its own options allow; with access only to current visual state, the agent cannot reliably imitate any agent which itself relies on more than just this information. An example of this phenomenon occurs when imitating the NewKrislet-AntiSimpleton clients, which are aware of their absolute position on the soccer field; where the original client reliably returns to a specific home position, the imitating agent has no such knowledge and therefore would have difficulty generalizing about object positions (though it may be able to approximate them somewhat using distances from static objects, e.g. flags).

- *Simplistic object-matching algorithm.* The current object-matching algorithm is a rough heuristic based loosely on the skiers-and-skis problem [50]. Since it considers distance to the player (as opposed to directly matching nearby objects) and uses a single numerical value (e.g. rather than a dimensional vector) the matching of objects is not optimal and thus, neither is the distance calculation. An improved matching algorithm would improve the overall performance results.

- *Single-layered architecture.* The direct association of inputs to outputs, in all

but the simplest clients, is not generally practical; this is why the layered-learning model is common in RoboCup client architectures (e.g. [70]). Our results agree with these findings. A more robust imitative agent should itself have a layered architecture with lower level skills (shooting, passing, move to position, etc.) from which it can generalize logical actions and behaviours. Only in doing so would such an agent be adequately equipped to emulate the more sophisticated RoboCup agents. The current single-layered architecture can only reasonably be expected to be successful on similarly-architected agents, of which there are few (and of limited practical use).

- *Simplistic stimulus-response approach.* Due to the above constraints the current agent is limited to a simple stimulus-response approach which directly links inputs to outputs. This can cause the agent to be prone to loops or "freezes" if the response does not have the desired effect on the environment (such as when the agent attempts to kick unsuccessfully, and becomes stuck since the world state remains unchanged).

- *Lack of internal state variables.* Observed agents must be completely deterministic and it is helpful if they are stateless ("stateful" agents select between pursuing one of several different goals at different times; this results in confused behaviour from the imitative agent which is unable to make this differentiation). Consider again the NewKrislet-AntiSimpleton agent, which uses internal states to decide when to wait in its home position, when to shoot for a goal, and when to return to its home. The stateless imitating agent attempts to perform all these goals simultaneously. Future work may address this by building more complex models of the observed agent's behaviour, e.g. with Hidden Markov

Models (Section 3.3.2).

- *Coverage.* The current implementation of the scene-based agent relies on foreknowledge of every possible situation; if there is no scene closely matching an encounter, the agent will not behave correctly. This results in the storage of many hundreds or thousands of scenes. In typical applications of software agents, the associated memory and storage requirements may be impractical or simply unacceptable.

- *Search speed.* As the number of stored scenes increases (or the complexity of the search algorithm increases) the time required to search the scene-space becomes an issue. Agents that take too long (over 100 ms, in the RoboCup environment) appear to be slow and exhibit "jerky" performance, which may be unacceptable. This shows the need to reduce the number of stored scenes by pruning or generalization.

- *No generalizations or learning (yet).* The current scene-recognition process is an example of instance-based learning, where training instances are stored for retrieval using the k-nearest-neighbor search. No new knowledge is being generated as a result of the scene-matching process. Ideally the agent should be able to learn even from this approach, producing trends, eventually rules, solidifying what the agent has learned over time. Additionally, the agent currently cannot look beyond one scene at a time and thus cannot recognize higher-level events or goals.

- *Not fully automated (yet).* There is still some human intervention required to select appropriate parameters for the proper results, such as object and action

weights, scene discretization table size (if used), and distance calculation algorithm selection. Currently, qualitative analysis is the most reliable measure of the imitative agent's success. The sample statistical information (kick accuracy) and the associated accuracy "spikes" in the bar graphs of Chapter 8 suggests that with the appropriate statistical measures to measure performance quantitatively, much of this process can eventually be automated.

9.3 Future Work

We are hopeful that many of the above issues may be addressed through future development and enhancement of the scene matching and recognition algorithms and through the further development of the scene-based RoboCup client. There are a number of specific enhancements that could be made to the current system, including the following:

- *Improved object matching algorithm.* Alternate methods for matching of objects can be explored, such as an implementation of Edmonds minimum weight perfect matching algorithm [49] for optimal object matching. An appropriate and effective object matching algorithm is critical to the performance of the scene recognition agent.

- *Detection of "stateless behaviour" in observed agents.* This could be determined by examining the observed data and considering whether any sets of scenes have the same inputs yet different outputs, i.e. $\exists x_1, x_2 | S(x_1) = S(x_2)$ but $R(x_1) \neq R(x_2)$. Large numbers of such scenes would suggest different states influencing the outputs.

- *A measure of scene distance beyond a single numerical value.* The current numerical distance computations are not distinct, since different combinations of objects in a given scene may each still result in the same final distance value. As such, it may be possible that scenes are calculated as "close" when they really are not. A solution to this may be to consider "distance" as a vector or an array of values, rather than a single numerical constant.

- *A set of qualitative evaluation metrics.* A better set of evaluation metrics would be helpful in automating some or all of the parameter adjustment process, reducing the current dependence on human observation and evaluation of the imitative agent. Current metrics for action accuracy do not necessarily reflect effective performance as a soccer-playing agent.

- *Object memories.* The recognition system currently does not provide support for a memory of previous states or objects previously seen but now out of sight. It may also be expanded to support projections in time of future states of objects given their current position and velocities. Some initial development has been taken (e.g. [31]).

- *Hierarchical scene storage, redundant scene elimination.* The current algorithms slow down dramatically (linearly with the number of total objects considered) as more object classes are considered or as the stored-scene file size increases. The discrete-cell scene structure supports simplifications of the scene database such as clustering many similar scenes (frequently adjacent) into one. Additionally, scenes can be stored in hierarchical format (such as a binary tree structure) to reduce search time. For example, the search may be optimized by sorting the

scenes by presence/absence of objects. Such a system was proposed in [76] but to do so requires the use of a priori knowledge (and the willingness to make assumptions about what types of situations may be considered similar).

- *Pattern mining within scenes.* There is a need to look beyond individual scenes, toward finding patterns or trends over sequences of scenes. An ideal scene-recognition system should be able to scan sequences of scenes and draw from them sets of general rules to follow (as in chronicles [17, 22]). The agent should also be able to trigger sequences of actions over consecutive time periods (e.g. to pass, to reach a certain region).

- *Incorporation of reinforcement learning approaches.* Rather than being completely guided by the low-level scene recognition process, the agent could monitor its own actions and generate rules describing the actions it takes. Given appropriate evaluation metrics, the agent could even learn to adjust its own object weights and other parameters to optimize them and improve its performance, whether in the field in real time, or in an offline training process (similar to the parameters of a neural network, but on a larger scale). This could result in a much more fully automated process for observation and imitation.

  Additionally, the outputs of other learning systems may be applied to scene learning or generation. For example, Paul Marlow's Classifier [51] can be used to experimentally learn the RoboCup boundary values (ball kickable, etc.), which could then be fed into the scene generator. Similarly, learned rules can be applied on action states, so that the agent can decide to "pass" as an action and have the decision logic determine how and where to best pass (like the SmartITAS system developed in [51]).

Finally, more experimentation should be undertaken, particularly to determine the effects of object and action weights (e.g. removing the restriction $w \in \{0, 1\}$) or to consider the effects of different discretized-region sizes other than $(5, 3)$.

9.4 Concluding Remarks

The scene recognition framework and algorithms described here represent some initial steps toward an automated process for observation and imitation of other agents. The experiments and results demonstrate its feasibility, at least on a small scale, and some (optimistic) direction is provided on enhancing the algorithms to improve their flexibility. There is room for much improvement and further experimentation. It is the author's hope that the incorporation of these techniques from the many different areas of machine learning and data mining can result in an improvement in the way software agents are developed and trained in the near future.

Bibliography

[1] Khurrum Ashfaque, Saurabh Bagrodia, Laurentiu Checiu, and Deryck Velasquez. *Manual Scene Generation.* SYSC 5103 Software Agents course project report, Carleton University, 2004.

[2] Jacky Baltes and Yong Joo Park. Comparison of several machine learning techniques in pursuit-evasion games. In *RoboCup-01: Robot Soccer World Cup V*, New York, 2002. Springer.

[3] Andraz Bezek. Modeling multiagent games using action graphs. In *In Proceedings of the AAMAS 2004 Workshop on Modeling Other agents from Observations (MOO 2004)*, pages 101–104, 2004.

[4] Serge Bibas, Marie-Odile Cordier, Philippe Dague, Christophe Dousson, Francois Levy, and Laurence Roze. Alarm driven supervision for telecommunication networks : I- Off-line scenario generation. *Annals of Telecommunications*, 51(9-10):493–500, 1996. CNET, France Telecom.

[5] John Black and Lina J. Karam. Automatic detection and extraction of perceptually significant visual features. In *Records 31st Asilomar Conference on Signals, Systems and Computers*, pages 315–319, 1997.

[6] Samuel R. Buss and Peter N. Yianilos. A bipartite matching approach to approximate string comparison and search. Technical report, NEC Research Institute, 1995.

[7] Juan P. Caraca-Valente and Ignacio Lopez-Chavarrias. Discovering similar patterns in time series. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 497–505. ACM Press, 2000.

[8] Sanjay Chandrasekharan, Babak Esfandiari, and Neal Arthorne. Using the robocup simulation environment to study the advantage of the environment contributing to cognition. Technical Report 2004-04, Carleton University Cognitive Science, 2004. Available from: `http://www.carleton.ca/iis/TechReports/files/2004-04.pdf` [cited 09/08/2005].

[9] Sanjay Chandrasekharan, Babak Esfandiari, and Neal Arthorne. Why soccer players yell: Using robocup to model the advantage of signaling. In M. C. Lovett, C. D. Schunn, C. Lebiere, and P. Munro, editors, *Proceedings of the 6th International Conference on Cognitive Modeling: ICCM 2004*, Pittsburg, PA, 2004.

[10] Mao Chen, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Itsuki Noda, Oliver Obst, Pat Riley, Timo Steffens, Yi Wang, and Xiang Yin. *RoboCup Soccer Server Manual*, 2002. Available from: `http://sourceforge.net/projects/sserver` [cited 09/08/2005].

[11] Ragini Choudhury, J. B. Srivastava, and Santanu Chaudhury. Reconstruction-based recognition of scenes with translationally repeated quadrics. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(6):617–632, 2001.

[12] Vitoria Robot Club. VitoriaRC simulation league team links. Available from: `http://www.dcc.fc.up.pt/~pribeiro/robocup/#visual` [cited 09/08/2005].

[13] A G Cohn and S M Hazarika. Qualitative spatial representation and reasoning: An overview. *Fundamenta Informaticae*, 46(1-2):1–29, 2001.

[14] Anthony Cohn, Derek R. Magee, Aphrodite Galata, David C. Hogg, and Shyamanta M. Hazarika. Towards an architecture for cognitive vision using qualitative spatio-temporal representations and abduction. In *Lecture Notes in Computer Science*, volume 2685, pages 232–248. Springer-Verlag, 2003.

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

[16] Klaus 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.

[17] C. Dousson and T. Vu Duong. Discovering chronicles with numerical time constraints from alarm logs for monitoring dynamic systems. In Thomas Dean, editor, *Sixteenth International Joint Conference on Artificial Intelligence*, pages 620–626, Stockholm, Sweden, 1999. Morgan Kaufmann.

[18] Christophe Dousson. Alarm driven supervision for telecommunication networks : II- On-line chronicle recognition. *Annals of Telecommunications*, 51(9-10):501–508, 1996. CNET, France Telecom.

[19] Christian Drücker, Sebastian Hübner, Ubbo Visser, and Hans-Georg Weland. "As Time Goes By" - using time series based decision tree induction to analyze the behaviour of opponent players. In *RoboCup 2001: Robot Soccer World Cup V*, pages 325–330, London, UK, 2002. Springer-Verlag.

[20] Mohammad El-Ramly, Eleni Stroulia, and Paul Sorenson. From run-time behavior to usage scenarios: an interaction-pattern mining approach. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 315–324. ACM Press, 2002.

[21] Babak Esfandiari. Software agent course website. Available from: `http://eureka.sce.carleton.ca:8080/cgi-bin/agentcourse.cgi` [cited 09/08/2005].

[22] Babak Esfandiari, Gilles Deflandre, Joel Quinqueton, and Christophe Dony. Agent-oriented techniques for network supervision. In *Ann. Telecommun 51, no 9-10*, page 521, 1996.

[23] Benoit Essiambre, David Tudino, and Chris Desmarais. *Calculating Weights for Nearest Neighbor Agent Emulation for RoboCup.* SYSC 5103 Software Agents course project report, Carleton University, 2004.

[24] The RoboCup Federation. Robocup official site. Available from: `http://www.robocup.org` [cited 09/08/2005].

[25] Adam Feldman and Tucker Balch. Modeling honey bee behavior for recognition using human trainable models. In *In Proceedings of the AAMAS 2004 Workshop on Modeling Other agents from Observations (MOO 2004)*, pages 17–24, 2004.

[26] Takeshi Fukase, Yuichi Kobayashi, Ryuichi Ueda, Takanobu Kawabe, and Tamio Arai. Real-time decision making under uncertainty of self-localization results. In *Proceedings of 2002 International RoboCup Symposium*, pages 375–383, 2002.

[27] S.R. Garner. WEKA: The waikato environment for knowledge analysis. In *Proceedings of the New Zealand Computer Science Research Students Conference*, pages 57–64, 1995.

[28] Christy Gnanapragasam, Rajan Balasubramaniam, and Satheeskumar Vaithilingam. *Agent 007: A RoboCup Client System for Software Agents.* SYSC 5103 Software Agents course project report, Carleton University, 2004.

[29] The RoboCup Soccer Simulator Maintenance Group. The robocup soccer simulator. Available from: `http://sserver.sourceforge.net` [cited 09/08/2005].

[30] Kwun Han and Manuela Veloso. Automated robot behavior recognition applied to robotic soccer. In John Hollerbach and Dan Koditschek, editors, *Robotics Research: the Ninth International Symposium*, pages 199–204. Springer-Verlag, London, 2000. Also in the Proceedings of IJCAI-99 Workshop on Team Behaviors and Plan Recognition.

[31] Shawn Henry, Chris Kafka, Ding Wei, and Fan Ping Zhou. *Enhanced Scenes Client: Zombies Ate My Agent.* SYSC 5103 Software Agents course project report, Carleton University, 2004.

[32] Timothy Huang and Frank Swenton. Teaching undergraduate software design in a liberal arts environment using robocup. In *ITiCSE '03: Proceedings of the 8th annual conference on Innovation and technology in computer science education*, pages 114–118, New York, NY, USA, 2003. ACM Press.

[33] Yoshiteru Ishida. A spatial recognition method for robots by symbolic processing: A puzzle ring solver. *International Workshop on Intelligent Robots and Systems IROS 91*, pages 502–507, 1991.

[34] Nico Jacobs, Kurt Driessens, and Luc De Raedt. Using ILP-systems for verification and validation of multi-agent systems. In *ILP '98: Proceedings of the 8th International Workshop on Inductive Logic Programming*, pages 145–154, London, UK, 1998. Springer-Verlag.

[35] Gal Kaminka and Dorit Avrahami. Symbolic behavior-recognition. In *In Proceedings of the AAMAS 2004 Workshop on Modeling Other agents from Observations (MOO 2004)*, pages 73–79, 2004.

[36] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The Robot World Cup Initiative. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 340–347, New York, 5–8, 1997. ACM Press.

[37] Jelle R. Kok and Nikos Vlassis. Mutual modeling of teammate behavior. Technical Report IAS-UVA-02-04, Computer Science Institute, University of Amsterdam, The Netherlands, 2002.

[38] Gregory Kuhlmann, Peter Stone, and Justin Lallinger. The UT Austin Villa 2003 champion simulator coach: A machine learning approach. In Daniele Nardi, Martin Riedmiller, and Claude Sammut, editors, *RoboCup-2004: Robot Soccer World Cup VIII*. Springer Verlag, Berlin, 2005. To appear.

[39] Kevin Lam and Gabriel Wainer. Modeling of maze-solving systems using Cell-DEVS. In *Proceedings of the 2003 Summer Simulation Multiconference*, 2003.

[40] Krzysztof Langner. The Krislet Java Client. Available from: `http://www.ida.liu.se/~frehe/RoboCup/Libs/libsv5xx.html#Krislet` [cited 09/08/2005].

[41] Krzysztof Langner. Roboplayer home page. Available from: `http://roboplayer.sourceforge.net/` [cited 09/08/2005].

[42] Antonietta Lanza, Donato Malerba, Francesca A. Lisi, Annalisa Appice, and Michelangelo Ceci. Generating logic descriptions for the automated interpretation of topographic maps. In D. Blostein and Y.-B. Kwon, editors, *Graphics Recognition: Algorithms and Applications*, volume 2390 of *Lecture Notes in Computer Science*, pages 193–203. Springer, Berlin, 2002.

[43] Jerome Lemaire. Use of a priori descriptions in a high-level language and management of the uncertainty in a scene recognition system. In *Proceedings of the International Conference on Pattern Recognition '96*, page 560, 1996.

[44] Jerome Lemaire and Olivier Le Moigne. Development of a scene recognition system with imprecise descriptions. In *International Conference on Image Processing '96*, page 979, 1996.

[45] Kristina Lerman and Aram Galstyan. Automatically modeling group behavior of simple agents. In *In Proceedings of the AAMAS 2004 Workshop on Modeling Other agents from Observations (MOO 2004)*, pages 49–55, 2004.

[46] Guiquan Liu, Xiaoping Chen, Xufa Wang, and Bo Zhang. On multi-agent collaborative planning and its application in Robocup. In *Proceedings of the 3rd World Congress on Intelligent Control and Automation*, page 184, June 28, 2000.

[47] Qi Liu, Tao Yue, Hui Shang, and Yi Qi. *Learning from Recorded Games Using Decision Tree.* SYSC 5103 Software Agents course project report, Carleton University, 2004.

[48] Therese Lowery. BEAM robotics website. Available from: `http://www.nis.lanl.gov/projects/robot/` [cited 09/08/2005].

[49] Michael Maguire and Luis Goddyn. Visual matching. Available from: `http://www.math.sfu.ca/~goddyn/Courseware/Visual_Matching.html` [cited 09/08/2005].

[50] Dinesh Manocha and Mark Foskey. COMP 122 - algorithms and analysis practice final, December 2001. Available from: `http://www.cs.unc.edu/~geom/TEACH/COMP122/PDF/p_final_soln.pdf` [cited 06/30/2005].

[51] Paul 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.

[52] Tohgoroh Matsui, Nobuhiro Inuzuka, and Hirohisa Seki. A proposal for inductive learning agent using first-order logic. In J. Cussens and A. Frisch, editors,

Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming, pages 180–193, 2000.

[53] Kurt Mehlhorn and Guido Schäfer. Implementation of o(nmlogn) weighted matchings in general graphs: the power of data structures. *J. Exp. Algorithmics*, 7:4, 2002.

[54] Stuart Middleton. Interface agents: A review of the field, 2001-2002. Available from: `http://www.ecs.soton.ac.uk/~sem99r/agent_survey.html` [cited 09/08/2005].

[55] Jeffrey P. Morrill. Distributed recognition of patterns in time series data. *Communications of the ACM*, 41(5):45–51, 1998.

[56] Aloke Tukul Mukherjee. Stripslet: a robocup-playing STRIPS planner based on krislet. Available from: `http://www.employees.org/~alokem/robocup-sw/stripslet.html` [cited 09/08/2005].

[57] Yohei Murakami, Toru Ishida, Tomoyuki Kawasoe, and Reiko Hishiyama. Scenario description for multi-agent simulation. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 369–376. ACM Press, 2003.

[58] Andreas G. Nie, Angelika Honemann, Andres Pegam, Collin Rogowski, Leonhard Hennig, Marco Diedrich, Philipp Hugelmeyer, Sean Buttinger, and Timo Steffens. The Osnabrueck RoboCup Agents Project. Technical report, Institute of Cognitive Science, Osnabrueck, 2001.

[59] B. J. Oommen and R. K. S. Loke. Pattern recognition of strings with substitutions, insertions, deletions and generalized transpositions. *Pattern Recognition*, 30:789–800, 1997.

[60] Luis Paulo Reis and Nuno Lau. FC Portugal team description: RoboCup 2000 simulation league champion. In *RoboCup 2000: Robot Soccer World Cup IV*, pages 29–40, London, UK, 2001. Springer-Verlag.

[61] Collin Rogowski. Model-based opponent modelling in domains beyond the prisoner's dilemma. In *In Proceedings of the AAMAS 2004 Workshop on Modeling Other agents from Observations (MOO 2004)*, pages 41–48, 2004.

[62] Michael Schroeder and Penny Noy. Multi-agent visualisation based on multivariate data. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 85–91, New York, NY, USA, 2001. ACM Press.

[63] Li Shi, Chen Jiang, Ye Zhen, and Sun Zengqi. Learning competition in robot soccer game based on an adapted neuro-fuzzy inference system. In *Proceedings of the 2001 IEEE International Symposium on Intelligent Control*, pages 195–199, Mexico City, Mexico, 2001.

[64] Li Shi, Yao Jinyi, Ye Zhen, and Sun Zengqi. Multiple rewards fuzzy reinforcement learning algorithm in robocup environment. In *Proceedings of the 2001 IEEE International Conference on Control Applications*, pages 317–322, Mexico City, Mexico, 2001.

[65] Katsunari Shibata. Spatial recognition model by extracting correlated information between vision and motion information using neural network. In *Proceedings of 1993 International Joint Conference on Neural Networks*, page 2536, 1993.

[66] Hans J.W. Spoelder, Luc Renambot, Desmond Germans, Henri E. Bal, and Frans C.A. Groen. Man multi-agent interaction in VR: a case study with RoboCup. In *IEEE Virtual Reality 2000 Conference*, page 291, 2000.

[67] Timo Steffens. Feature-based declarative opponent-modelling in multi-agent systems. Master's thesis, Institute of Cognitive Science Osnabruck, 2002.

[68] Frieder Stolzenburg, Oliver Obst, Jan Murray, and Bjorn Bremer. Spatial agents implemented in a logical expressible language. In Manuela Veloso, Enrico Pagello, and Hiroaki Kitano, editors, *RoboCup-99: Robot Soccer WorldCup III*, volume 1856. Springer, 2000.

[69] P. Stone, M. Veloso, and P. Riley. The CMUnited-98 champion simulator team. In Minoru Asada and Hiroaki Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*, Berlin, 1999. Springer Verlag.

[70] Peter Stone. Layered learning in multiagent systems. In *AAAI/IAAI*, page 819, 1997.

[71] Peter Stone. *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer.* MIT Press, 2000.

[72] Peter Stone, Patrick Riley, and Manuela Veloso. Defining and using ideal teammate and opponent agent models. In *Proceedings of the Twelfth Innovative Applications of AI Conference*, 2000.

[73] Frank Swenton. *NKClient Reference*, 2003. Available from: `http://bj.middlebury.edu/~huang/NKClientWeb/NKBrain.pdf` [cited 09/08/2005].

[74] Y. Takahashi and M. Asada. Vision-guided behavior acquisition of a mobile robot by multi-layered reinforcement learning. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2000.

[75] Andrew Vardy and R. Moller. Biologically plausible visual homing methods based on optical flow techniques. In *Connection Science, Special Issue: Navigation*, 2004.

[76] Alan Wai, Rachid Chreyh, and James Kelly. *RoboCup Project Report: Scene Learning on Others.* SYSC 5103 Software Agents course project report, Carleton University, 2004.

[77] Gerhard Weiss, editor. *Multiagent systems : a modern approach to distributed artificial intelligence.* MIT Press, Cambridge, Mass., 1999.

[78] Alexandre Winter and Chahab Nastar. Differential feature distribution maps for image segmentation and region queries in image databases. In *CBAIVL '99: Proceedings of the IEEE Workshop on Content-Based Access of Image and Video Libraries*, page 13, Washington, DC, USA, 1999. IEEE Computer Society.

[79] Ming Xie, Asif Muhammad, and Zhen-Eric Zhang. *Final Project Report: Zed Robocup Soccer Client.* SYSC 5103 Software Agents course project report, Carleton University, 2004.

[80] Xuming Xu, Shi Li, Zhen Ye, and Zeng Qi Sun. A survey: Robocup and the research. In *Proceedings of the 3rd World Congress on Intelligent Control and Automation*, page 207, June 28, 2000.

[81] A. Yamada, Y. Shirai, and J. Miura. Tracking players and a ball in video image sequence and estimating camera parameters for 3D interpretation of soccer games. In *Proceedings of the 16th International Conference on Pattern Recognition*, pages 303–306, 2002.

[82] Jinyi Yao, Jiang Chen, Yunpeng Cai, and Shi Li. Architecture of tsinghuaeolus. In *RoboCup 2001: Robot Soccer World Cup V*, pages 491–494, London, UK, 2002. Springer-Verlag.

[83] Jinyi Yao, Jiang Chen, and Zeng Qi Sun. An application in Robocup combining Q-learning with adversarial planning. In *Proceedings of the 4th World Congress on Intelligent Control and Automation*, page 496, June 10, 2002.

[84] Jinyi Yao, Lao Ni, Fan Yang, Yunpeng Cai, and Zengqi Sun. Technical solutions of tsinghuaeolus for robotic soccer. In *RoboCup 2003: Robot Soccer Woyld Cup VII*, pages 205–213. Springer, 2003.

[85] Masahiro Yokoi, Yuichi Kobayashi, Takeshi Fukase, Hideo Yuasa, and Tamio Arai. Learning self-localization with teaching system. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1211–1216, 2000.

[86] Kaizhi Yue. Generating interesting scenarios from system descriptions. In *Proceedings of the first international conference on Industrial and engineering applications of artificial intelligence and expert systems*, pages 212–218. ACM Press, 1988.

[87] Song Zhiwei, Zhang Bo, Chen Xiaoping, and Wang Xufa. Layered decision-making and planning in ShaoLing team. In *Proceedings of the 3rd World Congress on Intelligent Control and Automation*, 2000.

Appendix A

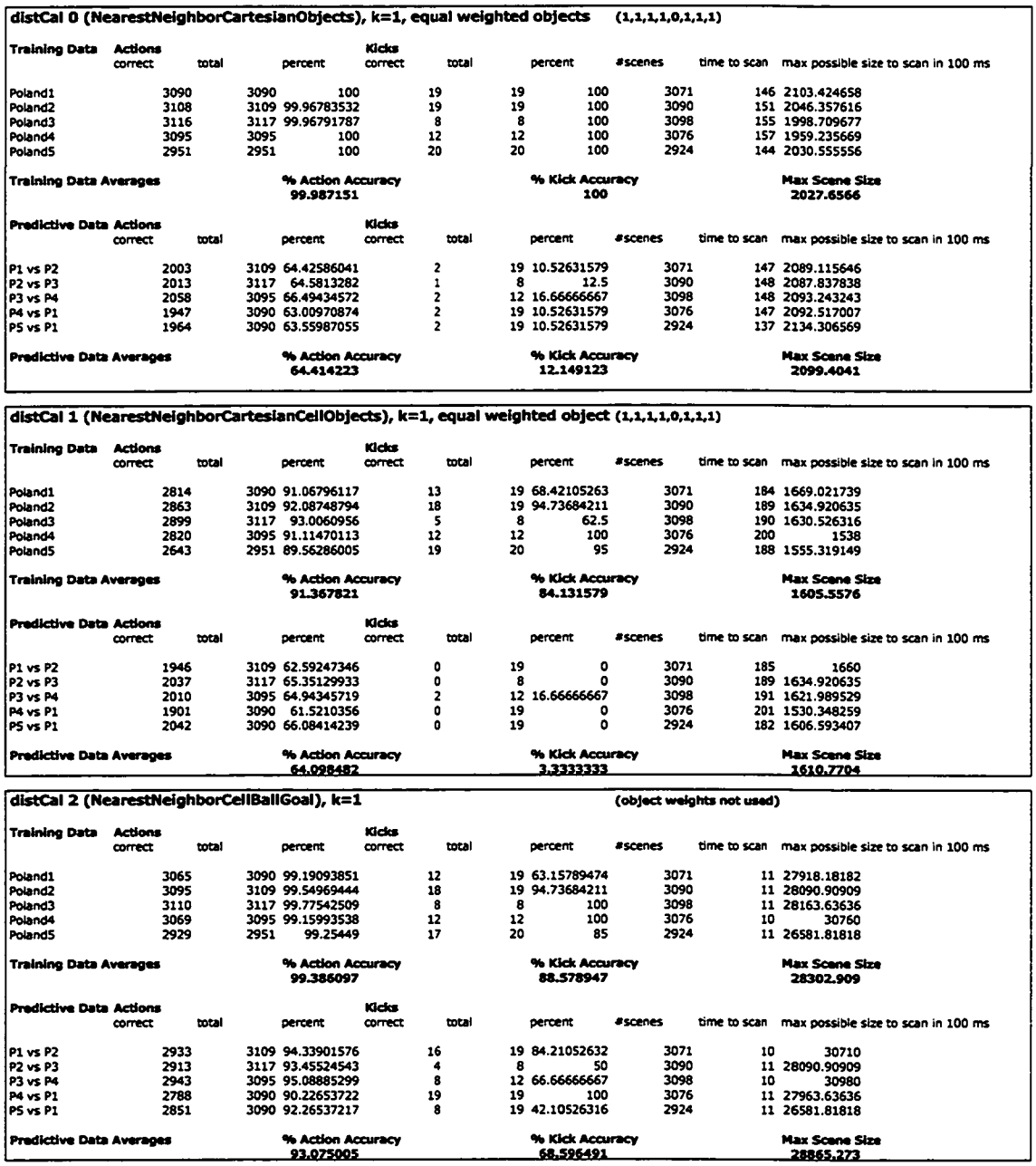
Sample Data Observations

> Errors using inadequate data are much less than those using no data at all.
>
> *Charles Babbage (1792-1871)*

As experimental results were generated, the raw data was formatted into a Microsoft Excel worksheet which calculated the percentages and averages of each trial. The average of the five trials was used in the final tables presented in Chapter 8.

The following pages show two sets of tables of data showing observations from the client with the Krislet scene files loaded. Figure A.1 shows the results as the DistanceCalculation algorithm is varied with all other parameters constant; Figure A.2 shows three results from variations of object weights.

The process was repeated for each of the three observed agents (Krislet, CMUnited, NewKrislet-AntiSimpleton Attacker).



distCal 0 (NearestNeighborCartesianObjects), k=1, equal weighted objects (1,1,1,1,0,1,1,1)

Training Data	Actions correct	total	percent	Kicks correct	total	percent	#scenes	time to scan	max possible size to scan in 100 ms
Poland1	3090	3090	100	19	19	100	3071	146	2103.424658
Poland2	3108	3109	99.96783532	19	19	100	3090	151	2046.357616
Poland3	3116	3117	99.96791787	8	8	100	3098	155	1998.709677
Poland4	3095	3095	100	12	12	100	3076	157	1959.235669
Poland5	2951	2951	100	20	20	100	2924	144	2030.555556

Training Data Averages	% Action Accuracy	% Kick Accuracy	Max Scene Size
	99.987151	100	2027.6566

Predictive Data	Actions correct	total	percent	Kicks correct	total	percent	#scenes	time to scan	max possible size to scan in 100 ms
P1 vs P2	2003	3109	64.42586041	2	19	10.52631579	3071	147	2089.115646
P2 vs P3	2013	3117	64.5813282	1	8	12.5	3090	148	2087.837838
P3 vs P4	2058	3095	66.49434572	2	12	16.66666667	3098	148	2093.243243
P4 vs P1	1947	3090	63.00970874	2	19	10.52631579	3076	147	2092.517007
P5 vs P1	1964	3090	63.55987055	2	19	10.52631579	2924	137	2134.306569

Predictive Data Averages	% Action Accuracy	% Kick Accuracy	Max Scene Size
	64.414223	12.149123	2099.4041

distCal 1 (NearestNeighborCartesianCellObjects), k=1, equal weighted object (1,1,1,1,0,1,1,1)

Training Data	Actions correct	total	percent	Kicks correct	total	percent	#scenes	time to scan	max possible size to scan in 100 ms
Poland1	2814	3090	91.06796117	13	19	68.42105263	3071	184	1669.021739
Poland2	2863	3109	92.08748794	18	19	94.73684211	3090	189	1634.920635
Poland3	2899	3117	93.0060956	5	8	62.5	3098	190	1630.526316
Poland4	2820	3095	91.11470113	12	12	100	3076	200	1538
Poland5	2643	2951	89.56286005	19	20	95	2924	188	1555.319149

Training Data Averages	% Action Accuracy	% Kick Accuracy	Max Scene Size
	91.367821	84.131579	1605.5576

Predictive Data	Actions correct	total	percent	Kicks correct	total	percent	#scenes	time to scan	max possible size to scan in 100 ms
P1 vs P2	1946	3109	62.59247346	0	19	0	3071	185	1660
P2 vs P3	2037	3117	65.35129933	0	8	0	3090	189	1634.920635
P3 vs P4	2010	3095	64.94345719	2	12	16.66666667	3098	191	1621.989529
P4 vs P1	1901	3090	61.5210356	0	19	0	3076	201	1530.348259
P5 vs P1	2042	3090	66.08414239	0	19	0	2924	182	1606.593407

Predictive Data Averages	% Action Accuracy	% Kick Accuracy	Max Scene Size
	64.098482	3.3333333	1610.7704

distCal 2 (NearestNeighborCellBallGoal), k=1 (object weights not used)

Training Data	Actions correct	total	percent	Kicks correct	total	percent	#scenes	time to scan	max possible size to scan in 100 ms
Poland1	3065	3090	99.19093851	12	19	63.15789474	3071	11	27918.18182
Poland2	3095	3109	99.54969444	18	19	94.73684211	3090	11	28090.90909
Poland3	3110	3117	99.77542509	8	8	100	3098	11	28163.63636
Poland4	3069	3095	99.15993538	12	12	100	3076	10	30760
Poland5	2929	2951	99.25449	17	20	85	2924	11	26581.81818

Training Data Averages	% Action Accuracy	% Kick Accuracy	Max Scene Size
	99.386097	88.578947	28302.909

Predictive Data	Actions correct	total	percent	Kicks correct	total	percent	#scenes	time to scan	max possible size to scan in 100 ms
P1 vs P2	2933	3109	94.33901576	16	19	84.21052632	3071	10	30710
P2 vs P3	2913	3117	93.45524543	4	8	50	3090	11	28090.90909
P3 vs P4	2943	3095	95.08885299	8	12	66.66666667	3098	10	30980
P4 vs P1	2788	3090	90.22653722	19	19	100	3076	11	27963.63636
P5 vs P1	2851	3090	92.26537217	8	19	42.10526316	2924	11	26581.81818

Predictive Data Averages	% Action Accuracy	% Kick Accuracy	Max Scene Size
	93.075005	68.596491	28865.273

Figure A.1: Sample Raw Observed Data from Krislet Imitation (page 1)

Object Weight Experiments

distCal 0 (NearestNeighborCartesianObjects), k=1 **ball only (1,0,0,0,0,0,0,0)**

Training Data	Actions correct	total	percent	Kicks correct	total	percent	#scenes	time to scan	max possible size to scan in 100 ms
Poland1	3077	3090	99.579288	13	19	68.4210526	3071	23	13352.1739
Poland2	3091	3109	99.4210357	19	19	100	3090	21	14714.2857
Poland3	3092	3117	99.1979467	8	8	100	3098	22	14081.8182
Poland4	3080	3095	99.5153473	12	12	100	3076	21	14647.619
Poland5	2931	2951	99.3222636	17	20	85	2924	20	14620

Training Data Averages	% Action Accuracy	% Kick Accuracy	Max Scene Size
	99.407176	90.684211	14283.179

Predictive Data	Actions correct	total	percent	Kicks correct	total	percent	#scenes	time to scan	max possible size to scan in 100 ms
P1 vs P2	3062	3109	98.4882599	9	19	47.3684211	3071	22	13959.0909
P2 vs P3	3068	3117	98.4279756	4	8	50	3090	22	14045.4545
P3 vs P4	3022	3095	97.641357	3	12	25	3098	22	14081.8182
P4 vs P1	3016	3090	97.605178	5	19	26.3157895	3076	22	13981.8182
P5 vs P1	3022	3090	97.7993528	5	19	26.3157895	2924	21	13923.8095

Predictive Data Averages	% Action Accuracy	% Kick Accuracy	Max Scene Size
	97.992425	35	13998.398

distCal 0 (NearestNeighborCartesianObjects), k=1 **ball + goal only (1,1,0,0,0,0,0,0)**

Training Data	Actions correct	total	percent	Kicks correct	total	percent	#scenes	time to scan	max possible size to scan in 100 ms
Poland1	3089	3090	99.9676375	19	19	100	3071	38	8081.57895
Poland2	3107	3109	99.9356706	19	19	100	3090	47	6574.46809
Poland3	3116	3117	99.9679179	8	8	100	3098	39	7943.58974
Poland4	3093	3095	99.9353796	12	12	100	3076	41	7502.43902
Poland5	2948	2951	99.8983395	20	20	100	2924	39	7497.4359

Training Data Averages	% Action Accuracy	% Kick Accuracy	Max Scene Size
	99.940989	100	7519.9023

Predictive Data	Actions correct	total	percent	Kicks correct	total	percent	#scenes	time to scan	max possible size to scan in 100 ms
P1 vs P2	2399	3109	77.1630749	3	19	15.7894737	3071	38	8081.57895
P2 vs P3	2309	3117	74.0776388	3	8	37.5	3090	39	7923.07692
P3 vs P4	2462	3095	79.5476575	0	12	0	3098	39	7943.58974
P4 vs P1	2261	3090	73.171521	5	19	26.3157895	3076	38	8094.73684
P5 vs P1	2343	3090	75.8252427	8	19	42.1052632	2924	36	8122.22222

Predictive Data Averages	% Action Accuracy	% Kick Accuracy	Max Scene Size
	75.957027	24.342105	8033.0409

distCal 0 (NearestNeighborCartesianObjects), k=1 **ball + goal + flag (1,1,1,0,0,0,0,0)**

Training Data	Actions correct	total	percent	Kicks correct	total	percent	#scenes	time to scan	max possible size to scan in 100 ms
Poland1	3090	3090	100	19	19	100	3071	83	3700
Poland2	3108	3109	99.9678353	19	19	100	3090	99	3121.21212
Poland3	3116	3117	99.9679179	8	8	100	3098	102	3037.2549
Poland4	3095	3095	100	12	12	100	3076	139	2212.94964
Poland5	2950	2951	99.9661132	20	20	100	2924	87	3360.91954

Training Data Averages	% Action Accuracy	% Kick Accuracy	Max Scene Size
	99.980373	100	3086.4672

Predictive Data	Actions correct	total	percent	Kicks correct	total	percent	#scenes	time to scan	max possible size to scan in 100 ms
P1 vs P2	2021	3109	65.0048247	1	19	5.26315789	3071	86	3570.93023
P2 vs P3	2096	3117	67.244145	0	8	0	3090	119	2596.63866
P3 vs P4	2140	3095	69.1437803	0	12	0	3098	120	2581.66667
P4 vs P1	2085	3090	67.4757282	0	19	0	3076	121	2542.14876
P5 vs P1	1950	3090	63.1067961	4	19	21.0526316	2924	111	2634.23423

Predictive Data Averages	% Action Accuracy	% Kick Accuracy	Max Scene Size
	66.395055	5.2631579	2785.1237

Figure A.2: Sample Raw Observed Data from Krislet Imitation (page 2)

Appendix B

Sample Scripts for Agent Validation Tests

I didn't think; I experimented.

Wilhelm Roentgen (German physicist, 1845-1923)

The following sections list the Unix shell scripts used to start the RoboCup server and to perform the validation testing on the KrisletScenes agent. This is of particular importance for supporting future work on this project.

B.1 RoboCup Environment Startup

```
# start up the server and the soccer monitor
# must be started in an X11 environment!
./rcssserver &
./rcssmonitor
```

B.2 Validation Test Script

```
# Test script for validation of NewKrislet-AntiSimpleton Attacker

echo %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
echo % Set 1 - distCal variations
echo %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

echo %%%%% distCal 0 NearestNeighborCartesianObjects
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1

echo %%%%% distCal 1 NearestNeighborCartesianCellObjects
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 1 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 1 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 1 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 1 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 1 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1

echo %%%%% distCal 2 NearestNeighborCellBallGoal
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 2 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 2 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 2 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 2 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 2 -numBest 1 -objectWeights 1/1/1/1/0/1/1/1

echo %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
echo % Set 2 - k and sceneSel variations
echo %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

echo %%%%% k=5 equal weighted vote
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 2 -numBest 5 -sceneSel 2
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 2 -numBest 5 -sceneSel 2
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 2 -numBest 5 -sceneSel 2
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 2 -numBest 5 -sceneSel 2
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 2 -numBest 5 -sceneSel 2

echo %%%%% k=15 equal weighted vote
```

```
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 2 -numBest 15 -sceneSel 2
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 2 -numBest 15 -sceneSel 2
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 2 -numBest 15 -sceneSel 2
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 2 -numBest 15 -sceneSel 2
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 2 -numBest 15 -sceneSel 2

echo %%%%% k=5 random vote
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 2 -numBest 5 -sceneSel 1
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 2 -numBest 5 -sceneSel 1
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 2 -numBest 5 -sceneSel 1
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 2 -numBest 5 -sceneSel 1
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 2 -numBest 5 -sceneSel 1

echo %%%%% k=15 random vote
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 2 -numBest 15 -sceneSel 1
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 2 -numBest 15 -sceneSel 1
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 2 -numBest 15 -sceneSel 1
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 2 -numBest 15 -sceneSel 1
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 2 -numBest 15 -sceneSel 1

echo %%%%% k=5 choose first valid
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 2 -numBest 5 -sceneSel 0
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 2 -numBest 5 -sceneSel 0
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 2 -numBest 5 -sceneSel 0
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 2 -numBest 5 -sceneSel 0
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 2 -numBest 5 -sceneSel 0

echo %%%%% k=15 choose first valid
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 2 -numBest 15 -sceneSel 0
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 2 -numBest 15 -sceneSel 0
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 2 -numBest 15 -sceneSel 0
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 2 -numBest 15 -sceneSel 0
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 2 -numBest 15 -sceneSel 0

echo %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
echo % Set 3 - Object Weight Experiments
echo %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

echo %%%%% ball only 1/0/0/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 0 -numBest 1 -objectWeights 1/0/0/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 0 -numBest 1 -objectWeights 1/0/0/0/0/0/0/0
```

```
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 0 -numBest 1 -objectWeights 1/0/0/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 0 -numBest 1 -objectWeights 1/0/0/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 0 -numBest 1 -objectWeights 1/0/0/0/0/0/0/0

echo %%%%% goal only 0/1/0/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 0 -numBest 1 -objectWeights 0/1/0/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 0 -numBest 1 -objectWeights 0/1/0/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 0 -numBest 1 -objectWeights 0/1/0/0/0/0/0/0

echo %%%%% flags only 0/0/1/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 0 -numBest 1 -objectWeights 0/0/1/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 0 -numBest 1 -objectWeights 0/0/1/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 0 -numBest 1 -objectWeights 0/0/1/0/0/0/0/0

echo %%%%% lines only 0/0/0/1/0/0/0/0
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 0 -numBest 1 -objectWeights 0/0/0/1/0/0/0/0
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 0 -numBest 1 -objectWeights 0/0/0/1/0/0/0/0
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 0 -numBest 1 -objectWeights 0/0/0/1/0/0/0/0

echo %%%%% allPlayers only 0/0/0/0/1/0/0/0
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 0 -numBest 1 -objectWeights 0/0/0/0/1/0/0/0
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 0 -numBest 1 -objectWeights 0/0/0/0/1/0/0/0
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 0 -numBest 1 -objectWeights 0/0/0/0/1/0/0/0

echo %%%%% team sorted players only 0/0/0/0/0/1/1/1
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 0 -numBest 1 -objectWeights 0/0/0/0/0/1/1/1
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 0 -numBest 1 -objectWeights 0/0/0/0/0/1/1/1
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 0 -numBest 1 -objectWeights 0/0/0/0/0/1/1/1

echo %%%%% ball+goal only 1/1/0/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 0 -numBest 1 -objectWeights 1/1/0/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 0 -numBest 1 -objectWeights 1/1/0/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 0 -numBest 1 -objectWeights 1/1/0/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 0 -numBest 1 -objectWeights 1/1/0/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 0 -numBest 1 -objectWeights 1/1/0/0/0/0/0/0

echo %%%%% ball+goal+flag only 1/1/1/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/0/0/0/0/0
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/0/0/0/0/0
```

```
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/0/0/0/0/0

echo %%%%% ball+goal+flag+line only 1/1/1/1/0/0/0/0
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/0/0/0/0
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/0/0/0/0
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/0/0/0/0
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/0/0/0/0
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/0/0/0/0

echo %%%%% ball+goal+flag+line+allPlayers 1/1/1/1/1/0/0/0
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/1/0/0/0
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/1/0/0/0
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/1/0/0/0
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/1/0/0/0
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 0 -numBest 1 -objectWeights 1/1/1/1/1/0/0/0

echo %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
echo % Set 4 - random control
echo %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

echo %%%%% distCal 3 Random
java visiontable.KrisletScenes -scene nkas1.scene -validate nkas2.scene -distCal 3 -numBest 1
java visiontable.KrisletScenes -scene nkas2.scene -validate nkas3.scene -distCal 3 -numBest 1
java visiontable.KrisletScenes -scene nkas3.scene -validate nkas4.scene -distCal 3 -numBest 1
java visiontable.KrisletScenes -scene nkas4.scene -validate nkas5.scene -distCal 3 -numBest 1
java visiontable.KrisletScenes -scene nkas5.scene -validate nkas1.scene -distCal 3 -numBest 1

echo %%%%% Done!
```

B.3 Example Execution Output

```
Running with parameters:
host:
port: 6000
team: Poland
scenes for training: nkas4.scene
sceneSel: VoteWeighted (2)
distCal: RandomDistance (3)
numBest: 1
object weights: [1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]
action weights: [1.0,1.0,1.0,1.0,1.0,1.0,1.0]
Running in data validation mode against data file nkas5.scene
Loading stored scene file nkas4.scene
Loading scenes from file nkas4.scene
Read 2246 scenes, table size (3,5)
Loading target scene file nkas5.scene
Loading scenes from file nkas5.scene
Read 2084 scenes, table size (3,5)
Beginning validation of 2084 scenes
Validation complete.
Compared 2084 scenes against database of 2246 scenes
Total action comparisons: 3550
Total correct actions: 2325(65.49296%)
Total KICK actions: 94
Total correct KICKs: 2(2.1276596%)
Average time taken: 0 mill. per scene
-----------------------------
```

Appendix C

Student Contributions

> It is important that students bring a certain ragamuffin, barefoot, irreverence to their studies; they are not here to worship what is known, but to question it.
>
> *The Ascent of Man*

As a graduate course project for *SYSC5103 - Software Agents* (Carleton University, Faculty of Engineering) [21], students were given the opportunity to test, develop and enhance the core scene recognition framework. A number of contributions resulted from these efforts, including improved calculation algorithms, bug fixes and feature enhancements. Students also provided input on testing procedures and evaluation metrics, some of which resulted in the design of the `Validator` class and its algorithm.

C.1 Improved Distance Calculation

The *NearestNeighborCellBallGoalDistance* calculation algorithm was created by a student group [31]. This calculation algorithm considers only the positions and angles of

the ball and the opposing team's goal. The distance calculation uses a combination of the distances between physical coordinates (equation 6.2) and the differences in angle between the two objects; it also employs a higher weighting of the y-coordinate (distance). The combined effect is an optimization of the distance calculation which improves the effect of imitating simple agents such as Krislet (as the following chapter describes in detail).

C.2 Manual Scene Generation

One student group developed a Java GUI interface for viewing and generating scenes [1]. The eventual development of a practical GUI (Figure C.1) may aid in the use of the scene representation format for storing higher level knowledge (such as plays, strategies, etc.) Since manual generation of scenes requires a great deal of human effort (by definition) it is beyond the scope of this thesis. The generator tool also provides a way to view scenes to evaluate for correctness and for testing or debugging purposes.

C.3 Experimental Enhancements

Other student groups explored other enhancements to the scene recognition architecture, including:

- calculating relative scene weights [23], a precursor for the inclusion of object-weightings in the current framework;
- experimentation using the scene framework on several existing RoboCup clients

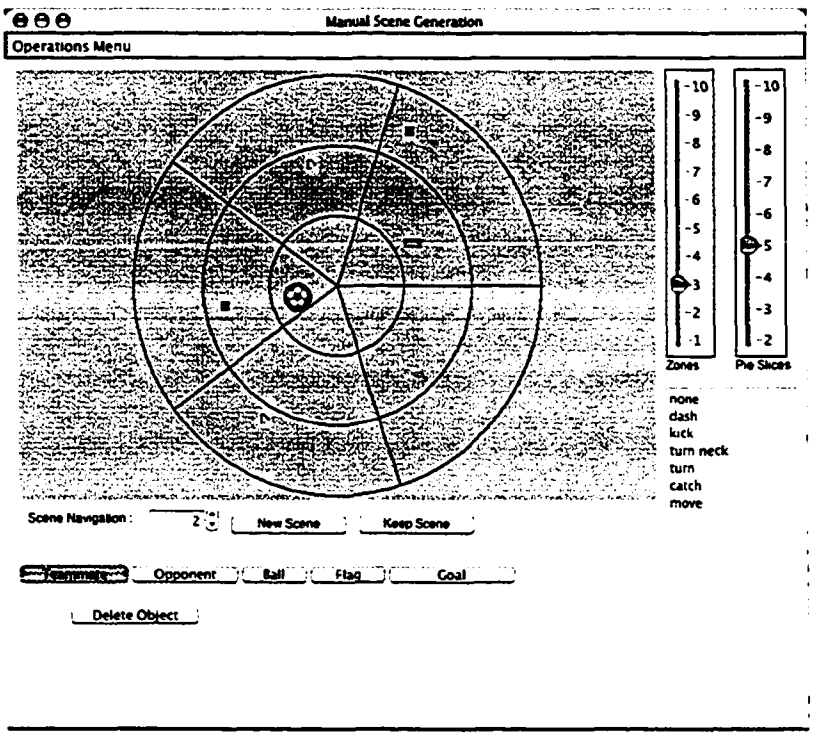


Figure C.1: Screen capture of the manual scene generation GUI [1].

[76], which included critiques and suggested improvements which led to a number of fixes in the current framework; the project report also contained suggestions for hierarchical storage of scenes which is considered for future work (Section 9.3);

- the impact of projections of object vectors to evaluate scenes based on the future (or past) positions of objects rather than simply the current positions [31]; student-obtained results appear promising and the process is considered for future work.

The relatively short length of time available for student development and testing precluded some direct contributions to the development of the framework, but all of the student group results were either merged into the current framework or under consideration for future development[1].

[1]Some development on enhancing the recognition performance is already being undertaken by students who participated in the course projects.