Coaching: Learning and Using Environment and Agent Models for Advice

Patrick Riley

CMU-CS-05-100

March 31, 2005

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Manuela Veloso, Chair
Tom Mitchell
Jack Mostow
Milind Tambe, University of Southern California

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was sponsored by the Department of the Interior-National Business Center under contract no. NBCHC030029, the US Air Force Research Laboratory under grant nos. F306020020549, F306029820135, and F306029720250, and the National Science Foundation through a Graduate Student Fellowship. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring institutions, the U.S. Government or any other entity.

Keywords: coaching, opponent modeling, machine learning, Markov Decision Process, simulated robot soccer

Abstract

Coaching is a relationship where one agent provides advice to another about how to act. This thesis explores a range of problems faced by an automated coach agent in providing advice to one or more automated advice-receiving agents. The coach's job is to help the agents perform as well as possible in their environment. We identify and address a set of technical challenges: How can the coach learn and use models of the environment? How should advice be adapted to the peculiarities of the advice receivers? How can opponents be modeled, and how can those models be used? How should advice be represented to be effectively used by a team? This thesis serves both to define the coaching problem and explore solutions to the challenges posed.

This thesis is inspired by a simulated robot soccer environment with a coach agent who can provide advice to a team in a standard language. This author developed, in collaboration with others, this coach environment and standard language as the thesis progressed. The experiments in this thesis represent the largest known empirical study in the simulated robot soccer environment. A predator-prey domain and and a moving maze environment are used for additional experimentation. All algorithms are implemented in at least one of these environments and empirical validation is performed.

In addition to the coach problem formulation and decompositions, the thesis makes several main technical contributions: (i) Several opponent model representations with associated learning algorithms, whose effectiveness in the robot soccer domain is demonstrated. (ii) A study of the effects and need for coach learning under various limitations of the advice receiver and communication bandwidth. (iii) The Multi-Agent Simple Temporal Network, a multi-agent plan representation which is refinement of a Simple Temporal Network, with an associated distributed plan execution algorithm. (iv) Algorithms for learning an abstract Markov Decision Process from external observations, a given state abstraction, and partial abstract action templates. The use of the learned MDP for advice is explored in various scenarios.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis is the conclusion of a full decade of schooling at Carnegie Mellon. Throughout my undergraduate and graduate years many people helped to guide me and support me during the journey that, eventually, led to this thesis.

I would like to thank my adviser Manuela Veloso for her guidance over the last seven years. She deserves credit for inspiring my continuing interest in AI. Both Manuela and Peter Stone provided my introduction to the simulated robot soccer environment, which, as should be apparent, had quite an impact on my future years. I would like to thank them both for our successful collaboration on the CMUnited teams. Peter also deserves thanks for the setplay data in Section 6.4.2.

I would also like to thank the members of my committee. Their perspectives have helped me lift my head from my own myopic perspective and I have gained a greater and deeper understanding of this research and research in general because of their insights.

Gal Kaminka deserves special thanks. His inspiration and hard work were major factors in the creation of the RoboCup coach competition. I would also like to thank the many members of the RoboCup community who helped with ideas, organization, and coding.

I never would have gotten to or through graduate school without the many wonderful people in my life outside of school. I feel very lucky to have such a large and supportive family. I would especially like to thank my sister Jenn for listening during both my triumphs and too often discussed complaints. My housemate George has been a great friend and co-perpetrator as we experimented with our "learning house." The coaching staff and girls at Sacred Heart volleyball provided me with a much needed and very rewarding bit of humanity during the abstractions of thesis research. I would also like to thank the Robinson's for their kindness, openness, and for providing me with an isolated tower where I had no choice but to finish my writing. Last, and in no possible way the least, Mary has brought much happiness into my life at a time when everything else was bringing me stress.

Thank you to everyone, both those who are mentioned here and those who are not, who helped me scale this particular mountain.

Notational Conventions

The following notational conventions are used throughout the text.

- $\mathbb{R}$ is the real numbers.
- $\mathbb{N}$ is the natural numbers, including 0.
- Sets are typeset in script, e.g. $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$
- Functions are typeset in capital letters and in italics, e.g. F, L.
- $\mathcal{P}$ is the powerset operator, so $\mathcal{P}(\mathcal{A})$ is the powerset of set $\mathcal{A}$.
- $\mathcal{A}^i$ is used to denote $\mathcal{A} \times \cdots \times \mathcal{A}$, the set of sequences of i elements of $\mathcal{A}$.
- $\mathcal{A}^*$ denotes $\cup_{i \in \mathbb{N}} \mathcal{A}^i$.
- For elements of a set given by a Cartesian product, $[\,]$ is used to specify components. For example, if $x \in \mathcal{A} \times \mathcal{B}$, then $x[1]$ is the $\mathcal{A}$ component and $x[2]$ is the $\mathcal{B}$ component.
- $x := y$ means that x is defined to be y.
- Tuples are enclosed in angle brackets, e.g. $\langle$a,b,c$\rangle$.
- URLs are typeset in monospaced font, e.g. `http://wolves.net`
- Advice language examples are typeset in monospaced font, e.g. `(pt ball)`

Chapter 1

Introduction

Multi-agent systems are increasingly becoming both more important and more complex. As the breadth, size, and importance of these systems grow, a better understanding of the various relationships between agents will be needed in order to design the most effective systems. This thesis considers one such relationship which has only recently begun to receive much attention: one agent acting as a coach for other agents.

1.1 Defining Coaching

While the word "coaching" is deliberately used to create associations with the rich human interactions with which we are familiar, we first clarify how coaching is defined and used in this thesis. We consider the primary properties of an agent coaching agents environment to be:

External, observing coach The coach is external to the team of agents performing in the environment. In particular, the coach's only effect on the environment is to communicate advice to the team of agents which are performing actions. In other words, the coach has no direct effect on the state of the environment. We assume that the agents are maintaining an internal state which is not accessible to either the other agents or the coach. This internal state can be affected by the advice of the coach, but the coach can not directly observe this effect. Further, we assume that the coach is able to observe (at least partially) the evolution of the environment in order to understand and evaluate the agents. In some cases, the coach can use observations to infer part of the internal state of the agents, but the coach can not directly observe that internal state.

Advice, not control The coach provides *advice* to the agents, and it does not control them. We assume advice recommends an action, a macro-action, or a set of goal states. The agents have some freedom to interpret the advice, and this interpretation will not be known *a priori* by the coach. Allowing advice interpretation is important because:

- Communication bandwidth restricts how specific advice can be. If the coach had infinite bandwidth with the receiver, the coach could explain its full understanding of the environment with all the good advice, exceptions to the advice, exceptions to the exceptions, ad nauseam. Advice in this thesis usually has the form of "In these states, do/don't do these actions." Indeed, if we consider "these states" to be an abstract state, we know that combining states which have similar, but not exactly the same, optimal actions can be more efficient for an agent to reason about [Dearden and Boutilier, 1997].
- Past research has suggested that in some cases the best performance can be obtained by an advice receiver only if it is able to refine advice [Maclin and Shavlik, 1996]. Indeed, with a coach agent that is not able to observe the internal state of the agents, advice will generally be abstract. Abstract advice creates a need for agents to refine and interpret the advice. By allowing agents to ignore some parts of the advice, we allow the agents to refine the advice given.

Access to past behavior logs One important task for coaching in human environments is the processing of past experience (or recordings of others' experiences) in order to provide information or insight to the advice receivers. This capability has a natural analogue in the agent coaching agents scenario and is one of the primary sources of information for our coach. Throughout the thesis, the coach will be analyzing logs of state transitions (in almost all cases without any information about the actions performed) in order to create various models.

Advice at execution, not training One task often performed by human coaches is to set up instructive training scenarios. Similarly, in many agent systems, the designers set up sub-tasks for learning by the agents before moving on the full task [Stone, 2000]. While creating such training scenarios may be an interesting task for a automated agent, it is not considered in this thesis in order to constrain the scope. The coach is providing advice about the execution of the task, not creating or providing advice for training scenarios.

Team goals One or more agents in the environment act to achieve given rewards. These rewards are team rewards which are the same for all agents on a team. The coach can therefore reason about one reward function for the team without having to consider interactions between differing goals. Environments with other agents are considered in this thesis, and in those environments the coach reasons about the behavior of the other agents but not about their goals.

For a coach in an environment with the properties above, the coaching problem is to provide advice to agents to improve their performance on the task. We present our decomposition of the overall coaching problem into sub-problems in Section 1.2.

Part of this thesis is in fact defining coaching for agent systems in an interesting way. In summary, our coach agents are external to the team they are coaching, provide advice to improve task execution, analyze logs of past behavior, and work with teams with shared rewards. While future research may want to broaden or narrow this definition of coaching, these properties specify the coaching problem for this thesis.

Part of the coaching problem is analyzing the past executions of agents. Control rule learning [e.g., Minton, 1988] and chunking [e.g., Laird et al., 1986b] also analyze execution traces in order to help a problem-solver perform better in the future. In this sense, control learning is similar to the coaching problem as addressed in this thesis. However, the robot-soccer-inspired coaching problem goes beyond control learning because the external nature of the coach allows for abstract advice to be generated without full knowledge of the internals of the advice receivers. This abstract advice can be operationalized by different types of agents.

Coaching brings together a number of different problem areas from AI. Intelligent Tutoring Systems (ITS) [e.g., Polson and Richardson, 1988] teach human students a task or skill such as reading or math. While they have similar inputs and outputs to our coach agents, these systems have important differences. ITSs are specialized to humans and they are generally used in environments where a complete and correct model of expert performance is available. Research in agent modeling [e.g., Carmel and Markovitch, 1998] aims to construct and use models of agent behavior based on observations. Advice generation is another use of such models. Using Markov Decision Processes as models of an environment for learning optimal action policies has been a fruitful technique [e.g., Puterman, 1994]. We will draw on this past work in creating models to be used by the coach in generating advice. Teamwork models such as STEAM [Tambe, 1997] define structured responsibilities and communication among agents in a team. While such models can define various leader roles, this thesis goes further in defining how a coach agent should act towards the rest of the team.

Combining these different ideas and areas provides fertile ground for exploring the properties of each. We draw on all of these areas to address the interesting and challenging problem of coaching.

1.2 Thesis Approach

Given the definition of the coaching problem, we can now state the thesis question:

Thesis Question: What algorithms can be used by an automated coach agent to provide advice to one or more agents in order to improve their performance?

The reader should note that the thesis question does *not* ask about the algorithms used by the advice receivers. While we have naturally had to address this question somewhat in order to do this thesis work (see especially Chapter 3, Section 6.2, and Carpenter et al. [2003]), this thesis focuses mainly on how advice is generated.

The coaching problem has a number of different sub-problems. This thesis serves to better define and break down the problems addressed. Through our insights into the various sub-problems, we open up future work in all areas.

In short, we break the overall thesis question into several sub-questions.

- How does the coach gain knowledge from observations and behavior logs?

  The coach processes observations of the environment and/or the agents to produce advice. These observations must have sufficient information to allow the coach to learn. Our coach learns models of teammate and opponent agents as well as models of the environment.

- How can models learned from observation be used to determine desired actions for agents?

  A model by itself does not produce advice. Rather, the coach must use the model to determine some desired ways of acting for the agents. These action policies can then be translated to advice. We answer this question in a variety of ways, including planning a response to predicted opponent actions, imitating other successful agents, and solving for a universal plan.

- Once the coach has desired actions for the team, how does the coach adapt advice to the agent abilities?

  Ideally, a coach should be able to work with a variety of agents with different abilities. Especially as the complexity of an environment grows, agents can be more or less effective at various tasks given physical or behavioral differences. We explore how a coach can learn about the advice receiving agents in order to provide the most effective advice.

- What format does advice take?

  The advice language constrains what kind of advice the coach can give and is consequently an important choice in the design of a system with a coach. In addition to using the standard advice language of the simulated robot soccer environment, we introduce a plan representation for use by distributed agents to perform coordinated execution.

Figure 1.1 overviews the entire approach contributed in this thesis. The figure shows the main building blocks and the chapters which address each. This figure represents the integration of the thesis at the conceptual level, as not all subsystems are integrated into one agent.

The coach's overall inputs and outputs are exactly as defined in the problem statement above (Section 1.1). Namely, the only output from the coach is advice to the agents and the inputs are observations from the environment and logs of past behavior.

Then, there are two primary paths to producing advice. One path deals with learning and using models of the environment. From logs of past behavior, the coach learns a model that captures not only the physical processes of the environment, but also the effects of the possible actions of the agents. This learning is done from logs that are series of states and do not include information about the actions performed (which are not directly observable by the external coach). Based on this model, the coach can solve for optimal policies. The coach then has a set of desired behaviors for the agents. This process is described in Chapter 4.

The other path deals with understanding particular opponents. Where the environment model intends to capture all the possible actions and processes in the environment, opponent models are intended to capture how particular opponent agents behave. The models are either given by an expert (as in Section 5.1) or learned from past observation data (as in Section 5.2). If the opponent is known, the appropriate opponent model can simply be used. Otherwise, the coach can use the current observations to select one of the opponent models from a given set (see Section 5.1.2). Online, model selection is done instead of model learning because there may not be enough data to learn a new model from scratch.

In some cases, using the opponent model is easy. If you are playing tag[1] and you

[1]The game of "tag," also known as catching, caçadinhas, and many other names, is a typical children's game. One person who is "it" runs around trying to touch, or "tag" someone who then becomes "it."

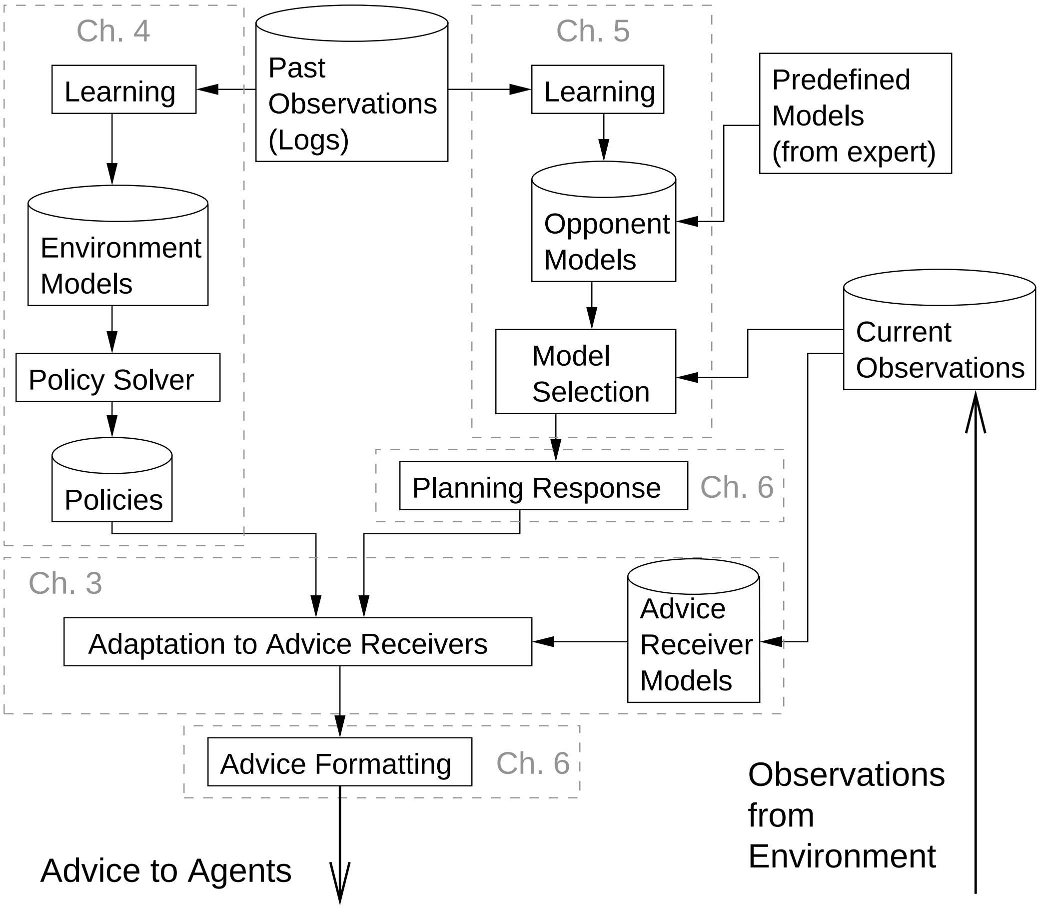


Figure 1.1: High level overview of the coach's advice generation process. The processes and data are marked with the thesis chapters that will discuss these in more detail.

know your opponent always turns left instead of right, it is easy to decide what to do. In other cases, significant computational effort must be applied to determine how to best respond to the predicted actions of the opponent. We present one planning approach to taking advantage of the opponent's predicted behavior (Section 6.3). As in the other path of Figure 1.1, the coach computes desired behaviors for the agents.

Given these desired behaviors, the coach must then consider how to best get the agents to behave as desired. The agents may have limitations that are unknown to the coach or the agents may be more or less effective at performing the actions suggested by the coach. The coach constructs and uses models of the agents receiving the advice in order to adapt the advice given. In Chapter 3, we present learning algorithms for the coach to do this adaptation.

At this point, the coach still has to express the advice in a language that the agents understand. This advice formatting step may be trivial if the output from the earlier processes closely matches the format of the advice language. However, the coach can encode additional information in the advice to make it easier for the agents to operationalize. This encoding can be especially useful when the advice is to be used by multiple agents simultaneously. This thesis introduces one representation for advice to be used by distributed agents, as well as algorithms to produce such advice. The representation and execution and generation algorithms are described in Chapter 6.

To this point, this overview has highlighted one dimension along which our various coaching algorithms vary, namely the source of advice. Several other important dimensions for describing coaching algorithms in this thesis:

Online vs. offline learning A coach can study past executions or environment models in order to produce advice for the agents. Alternatively, the coach could observe the agents' actual execution and provide advice as seems necessary to improve the performance while it is going on. The learning of environment models (Chapter 4) and the opponent team models (Section 5.2) exclusively use offline learning. The adaptation to advice receivers (Chapter 3), the selection of opponent models (Section 5.1), and the planning response (Section 6.3) are all performed online.

One-time vs. occasional vs. continual advice This dimension expresses how often the coach provides advice to the agents. The offline learning techniques all produce advice which is given to the receivers at the beginning of execution. In the agent adaptation learning (Chapter 3), we explore both continual and occasional advice. The planning (Chapter 6) using matched opponent models (Section 5.1) provides occasional advice.

Advice as actions vs. macro-actions vs. plans We use several different advice representations in this thesis. In the adaptation to advice receivers (Chapter 3), the advice takes the form of recommended base level actions. In the planning (Chapter 6), sequences of coordinated movements are provided to the team as advice. In all other work, we use the CLang advice language (described in Section 2.2.2), where the recommended actions are macro-actions that take several steps for the agents to implement.

We use several experimental environments throughout the thesis. We use some environments to isolate our sub-questions for more thorough study. The full details of all environments can be found in Chapter 2, but briefly, the environments are:

Predator-Prey We use this grid-based world with a single advice receiving agent to explore how a coach can adapt its advice to the abilities of the advice receiving agent.

Simulated Robot Soccer Experiments in the full game address all of the sub-questions above except learning about and adapting to the advice receivers. We also construct a sub-game to study the learning of environment models.

RCSSMaze We created this maze environment with dynamic walls to investigate the learning of environment models and clearly demonstrate the impact of the coach's advice.

1.3 Contributions

In addition to defining and decomposing the coaching problem as discussed above, this thesis makes several technical contributions:

- Several opponent model representations, learning algorithms for those models, and advice generation algorithms using those models. The use of these models is experimentally shown in a robot soccer domain. (Chapter 5)
- An empirical study of adapting advice in a predator-prey environment. Variations in the abilities of the advice receivers and the communication bandwidth are also explored. (Chapter 3)
- Multi-Agent Simple Temporal Networks (MASTNs): A novel multi-agent plan representation and accompanying execution algorithm. MASTNs are one answer to the

question of how advice should be structured so that it can be used efficiently by the receiving agents. (Chapter 6)

- An algorithm for learning an abstract Markov Decision Process from external observations, a given state abstraction, and partial abstract action templates. The learning and use are experimentally explored in a variety of scenarios. (Chapter 4)

- Largest empirical study of coaching in simulated robot soccer. (Chapters 4, 5, and 6)

1.4 Coaching in Other Domains

Since much of this thesis presents coaching techniques in the context of particular domains, especially simulated robot soccer, we would like to provides examples of how coaching could be applied in other domains. The coaching questions we present are more general than the domains presented in the thesis. Therefore, in this section we discuss how coaching could be applied to two other domains: the DARPA SDR Indoor Reconnaissance Demonstration [DARPA, 2002] and RoboCup Rescue [RoboCup Rescue].

1.4.1 Indoor Reconnaissance

The Defense Advanced Research Project Agency (DARPA) set out a challenge to use 100 robots semi-autonomously in an indoor environment to perform mapping, search, and intruder detection [DARPA, 2002]. This description of the problem and how coaching could be applied is based on the Centibots from SRI [Konolige et al., 2004].

There are three phases to the problem. In the first, a small number (1–3) of robots are placed in the previously unknown indoor environment. They are required to autonomously map the environment and are evaluated both on speed and accuracy of the resulting map. This map is then used for the subsequent phases.

In the second phase, a large number of robots (approximately 100) are used to locate a prespecified object of interest (OOI) which can be distinguished by its shape and color. The goal is to minimize both the time to find the OOI and the number of false positives.

In the last phase, the robots are required to guard the OOI and report when and if human intruders enter the space. A single human operator outside of the space to be guarded can interact with the robot team. There are two tasks in this phase: establishing a communication backbone and performing the sentry operations. The robots can only communicate with nearby robots, so a communication backbone is needed so that all robots can communicate with each other, especially when an intrusion is detected.

The simultaneous localization and mapping problem for phase one is a well established and studied problem, so we will instead focus on phases two and three.

First we consider the search problem of phase two. A software coach agent could exist on a computer in the central command also occupied by the single human operator. Given that the robots already maintain a communication backbone to keep the operator up to date on what is going on, the coach could use observation data supplied by the robots.

To see how coaching could be applied to this problem, we consider the four coaching sub-questions given above:

- How does the coach gain knowledge from observations and behavior logs?

  This question implicitly asks what form observations take. For this task, the coach will be interested in both possible locations of the OOI and areas which have been explored by each robot. From this information and the map constructed, the coach should be able to construct a model of the environment which consists of how well or how many times various areas have been scanned for the OOI. In order words, the coach could serve to merge the information provided by the distributed agents.

  In order to accomplish this, the coach will also need to have models of the sensation and action capabilities of the agents. If the higher level actions of the robots, such as scanning in a circle and detecting an object are fixed, the coach will still need to model the effectiveness of these actions in order to determine how well the environment has been scanned.

- How can models learned from observation be used to determine desired actions for agents?

  The Centibots implementation already includes a spatial reasoner and dispatcher to try to efficiently calculate spatial assignments for the robots. A crucial part of this reasoning is a set of cost functions and associated weights. The coach's model of how much and how thoroughly the various parts of the environment have been explored could be used to adjust these weights or cost functions as exploration happens.

- Once the coach has desired actions for the team, how does the coach adapt advice to the agent abilities?

  An important issue with so many robots is dealing with failures. The probability of no failures decreases exponentially with the number of robots and further, a robot may not even be aware when it is failing. If the coach is constructing models of the action efficacy of the robots, as suggested by the first question above, deviations from the expected behavior given by these models could be used to detect oncoming or current robot failures.

Further, if the robots are heterogeneous, and especially if they are designed or created by different groups then it will be important for a coach to understand the differences between the robots. The heterogeneous robots may move with different speeds and have different detection abilities.

- What format does advice take?

  The coach can indicate positions or areas of the map for each of the robots, or groups to robots, to explore. Since all robots are assumed to have the global map (from phase one) and properly localize, the coach can communicate in global coordinates on the shared map. This information could alternatively be processed by the spatial reasoner and dispatcher to modify the cost functions (as discussed in the second question above).

Now consider the intrusion detection problem of phase three. This problem is an even better match for coaching as described in this thesis as there exists an adversary (the intruder) that the team must deal with.

- How does the coach gain knowledge from observations and behavior logs?

  The Centibots implementation performs the sentry operation by spreading out the robots to monitor as much of the space as possible. However, this approach leaves significant room for improvement by modeling and prediction of possible locations and routes of the intruder. These models would predict, probabilistically, how and where the intruder would move through the space. If the robot sentries provide information to the coach about the areas monitored, the coach can effectively accumulate this information to understand the global monitoring situation.

- How can models learned from observation be used to determine desired actions for agents?

  A set of possible opponent models could be given to or learned by the system (similar to what is done in Section 5.1). Monitoring efforts could then be focused on likely positions from what is considered the most likely model or by using all models simultaneously.

  In addition, if the intruder is responding to the positions of the robots (as seems likely), it may be possible to plan a response (such as in Section 6.3) that effectively leads the intruder into an area where detection is almost certain. For example, if monitoring robots leave an area, the intruder may be more likely to move into this space to attempt to escape detection. Additional monitors could them be deployed in the correct area.

- Once the coach has desired actions for the team, how does the coach adapt advice to the agent abilities?

  This question has a similar answer as in phase two. The coach can construct models of agents which predict the efficacy of the actions. Violated predictions can mean a robot failure and better working robots can be deployed at more important locations. Further, as described above, models of teammate robots will be important as the team becomes more heterogeneous.

- What format does advice take?

  As should be clear at this point, the coach will be recommending areas to be monitored more or less by the robots. The spatial reasoner can interpret this advice as changes in weights for different areas.

1.4.2 RoboCup Rescue

RoboCup Rescue [RoboCup Rescue] is an international research project to promote the study of large heterogeneous teams in the socially important task of disaster rescue. This section considers the simulation system of RoboCup Rescue as an agent environment that could benefit from coaching. This discussion of the structure of the environment is drawn primarily from Morimoto [2002].

The RoboCup Rescue simulation environment simulates a city area after an earthquake. Buildings are collapsing, fires are starting and spreading, and civilians are buried, injured, and fleeing. There are three primary types of agents:

Ambulance These agents control an ambulance which is able to unbury people and carry them to rescue centers where they can be healed. The ambulance agents can broadcast communicate with each other and are coordinated by an ambulance center.

Fire Fighters These agents can extinguish fires, though it is possible that once a fire has gotten large, it will be more difficult or impossible for fire fighting agents to put it out. The fire fighting agents can broadcast communicate with each other and are coordinated by a fire station.

Police Police agents are responsible for clearing wreckage on a road. They can broadcast communicate with each other and are coordinated by a police office.

In addition, the three coordination centers can communicate with each other. The agents' goal is to reduce both loss of human life and destruction to the city.

To see how coaching could be applied to this problem, we consider the four coaching sub-questions given above.

- How does the coach gain knowledge from observations and behavior logs?

  One of the most important problems in the domain is the allocation of rescue agents to the various problems in the disaster scenarios. An important component of this reasoning could be abstract models of the effectiveness of agent(s) in addressing particular problems. For example, how many firefighters would be needed to put out a particular fire? How long would it take to clear a road given some number of policemen? How long will it take to get from point A to point B given the road and crowd situation? Such abstract models of the environment (including possible agent actions) are exactly what are learned in Chapter 4.

- How can models learned from observation be used to determine desired actions for agents?

  Using the abstract models above, the coach agent will still need to search for a good agent to task allocation. As part of the search process, the models can be used to predict future world and agent states.

- Once the coach has desired actions for the team, how does the coach adapt advice to the agent abilities?

  For most of the agents, the action space is fairly simple. The road clearing and rescue (i.e. unbury) actions, for example, take no parameters besides what road/human to act on. This means that there is probably little variation in the abilities of the agents to perform this task. However, the fire fighters have significantly more variation in the action space, having to choose how many hoses, how much water, and what angle to use. It the fire fighting agents are heterogeneous, the coach agent could usefully learn about the effectiveness of the various agents and assign agent to the fires for which they would be most effective.

- What format does advice take?

  The coach will be providing advice about which areas or disasters (collapsed buildings, fires, and road blockages) are most important to deal with. Additionally, the coach may need to provide coordinated plans for multiple agents (similar to the ones in Section 6.2) if there are important dependencies. For example, the fire fighters may need to move to a road at the same time that the police clear it.

1.4.3 Summary

This section has discussed how some of the coaching ideas discussed later in the thesis could be applied to two other domains, indoor reconnaissance and RoboCup Rescue. While different model representations and uses were suggested for the different problems, the idea of a coach analyzing the observations of the world to provide advice to be interpreted by the agents fit into both of the domains.

1.5 Document Outline

This section briefly summarizes the content of each of the following chapters.

Chapter 2 presents technical details of all of the domains used in the thesis and discusses the relevant properties of each, along with their differences. This chapter includes a brief discussion of the CLang advice language used in much of the thesis.

Chapter 3 discusses a number of algorithms for a coach to learn about the limitations of an advice receiving agent. A thorough empirical study in a predator-prey domain explores the use of different learning algorithms, agent limitations, and communication bandwidth.

Chapter 4 presents a set of algorithms to learn a model of an environment from observed state traces. The model is used to generate advice and is empirically tested in several variants of simulated robot soccer and in our own maze environment.

Chapter 5 discusses how opponent models can be learned and used to provide advice to agents. Experimental results in simulated robot soccer are shown.

Chapter 6 introduces Multi-Agent Simple Temporal Networks, a plan representation for distributed execution, and an associated plan execution algorithm. Empirical results are shown in simulated robot soccer.

Chapter 7 discusses the relation of this thesis to previous work in a number of different areas.

Chapter 8 concludes and provides directions for future work.

Appendix A gives additional details of the the CLang, the standard coach language for the simulated robot soccer community. Several extended examples are also presented. CLang is used as the advice language for much of the thesis.

Appendix B presents details of the action templates needed for the environment model learning in Chapter 4. The action templates for both simulated robot soccer and our maze environment are presented.

Appendix C gives details on getting the source code and data files from the thesis.

Appendix D presents our preliminary work on formal models of coaching.

Chapter 2

Environments

A variety of environments were used for both development and empirical validation of the techniques discussed in this thesis. This chapter discusses the relevant technical details of all of the domains used. The primary purpose of this chapter is to situate the environments in the context of why we used them.

Environments of varying complexity are needed to demonstrate and explore the properties of the various coaching algorithms presented. The first environment we discuss is a predator-prey environment. Because this environment is small enough that all optimal policies can be found and fully described, it allows for very thorough experimentation.

On the opposite end of the complexity spectrum is a simulated robot soccer environment, which is the main motivating environment for the thesis. A large community works in this rich, multi-agent, adversarial environment. Coach competitions have been run at the annual international RoboCup events since 2001 and teams are increasingly considering how to use a coach to improve their performance.

Lastly, we have a maze environment, called RCSSMaze, which we created. It preserves some of the complexity of the full soccer environment while allowing us to better control the environment and remove some of the unknowns. This control allows us to run more controlled experiments.

Table 2.1 summarizes some of the relevant properties of these environments. These properties will be further explained in the following sections.

	Predator-Prey	Simulated Soccer	RCSSMaze
States and actions	discrete	continuous	continuous
Observability	full	partial	partial
Multi-Agent/Adversarial	no	yes	no
Level of advice actions	lowest level	macro-actions	macro-actions
Optimal policy known	yes	no	probably
Advice frequency	variable	periodic	one time
Limited agent(s)	Yes, known	Yes, but unknown	Yes, but unknown

Table 2.1: Properties of the environments used in the thesis

2.1 Predator-Prey

State and action spaces : discrete
Observability : full
Multi-Agent/Adversarial : no
Level of advice actions : lowest level actions
Optimal policy known : yes
Advice frequency : variable (and configurable)
Limited agent(s) : Yes, and known exactly

The simplest environment we use is a version of the classic predator-prey. The simulation is built upon the SPADES simulation system [Riley and Riley, 2003, Riley, 2003], which provides a basic event based simulation framework. One predator and two prey agents move on a discrete 6x6 grid world. The agents have 5 actions: stay still, move north, south, east, or west. There are virtual walls on the outside of the grid so if an agent tries to move outside the grid, it stays where it is. An example of a world state and the possible actions is shown in Figure 2.1. The rabbits represent prey and the robot represents the predator.

The predator agent's goal is to capture at least one prey by being on the same square as it. The world is discrete, all agents have a global view of the world, and all moves take place simultaneously. Throughout our experiments, the coach's job is to advise the predator agent. Advice will be in the form of a recommended action, either to perform right now or at a given state in the future.

The actions of each agent are deterministic. If the predator uses the north action, it will always move north one square. However, the prey move randomly and therefore act as randomizers for the world, rather than learning agents which need to be handled.

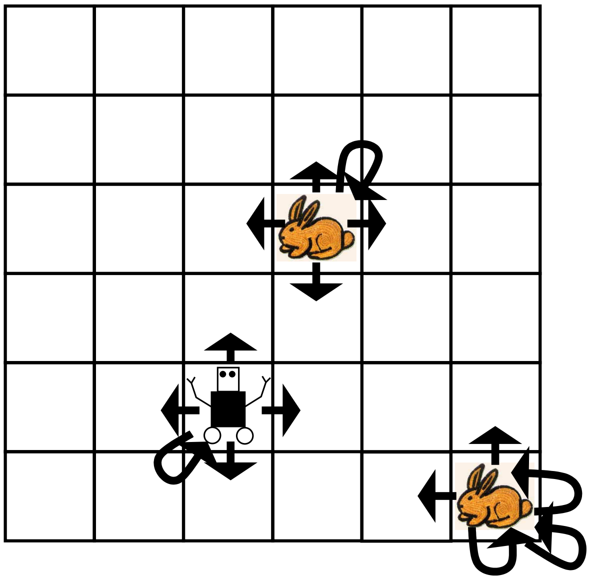


Figure 2.1: An example state in the predator-prey environment. The rabbits represent prey and the robot represents the predator. The arrows represent the possible actions (and their effects).

Therefore, when considering only the actions taken by the predator, the environment is non-deterministic.

The world operates episodically, with an episode ending when at least one prey is captured; the predator can simultaneously capture both prey if all three agents occupy the same square. The predator and coach receive a reward of 100 for each prey captured and a reward of -1 every step that is taken. The predator tries to maximize undiscounted reward per episode.

The state space of the world is $36^3 = 46656$ (the predator location, prey 1 location, prey 2 location), though any state which is a capture for the predator (2556 states) has no actions so no learning need take place. After a capture, all agents are placed randomly on the grid.

Because of the size of the environment, standard exact MDP solving techniques can be used [Puterman, 1994]. Approximately 20% of the states in this environment have multiple optimal actions. The optimal average reward per step is approximately 17, which means on average between 5 and 6 steps to capture a prey. Note that since future rewards are not discounted and when computing values we only consider rewards in this episode, the average reward per step reflects the value of a policy.

Knowing exactly the optimal policies allow us to do thorough experimentation and know exactly how close to optimal we are getting. Further, we can impose limitations on the agent and know exactly what effect those limitations will have on the optimal policies. These properties make this environment a good one to study issues of the coach adapting to the advice receiver.

2.2 Simulated Robot Soccer

State and action spaces : continuous
Observability : partial
Multi-Agent/Adversarial : yes
Level of advice actions : macro-actions
Optimal policy known : no
Advice frequency : periodic
Limited agent(s) : Yes, and limits unknown

The Soccer Server System [Noda et al., 1998] as used in RoboCup [Kitano et al., 1997, RoboCup] is the primary experimental domain for this thesis. This simulated robot soccer environment is a rich, multi-agent, adversarial environment. It is a compelling environment for the study of coaching for two reasons. First, a large number of teams have been created by a variety of research groups, giving considerable breadth to the strategy space in use by teams. Second, the CLang advice language was created by the community (including myself) as a standard language so that coaches could effectively talk to and work with teams from other research groups.

The size and complexity make this a challenging yet appealing environment for the coach. Much of the work in this thesis uses this domain. The learning and use of both environment and agent models in a variety of ways will be explained in the later chapters.

For additional information besides what is contained in this section, see the Soccer Server Manual [Chen et al., 2001] or the home page at `http://sserver.sf.net`.

2.2.1 Basic Operation

The Soccer Server System is a server-client system that simulates soccer between distributed agents. Clients communicate using a standard network protocol with well-defined actions. The server keeps track of the current state of the objects in the world, executes the actions requested by the clients, and periodically sends each agent noisy, incomplete information about the world. Agents receive noisy information about the direction and distance of objects on the field (the ball, players, goals, etc.); information is provided only for objects in the field of vision of the agent. A screen shot is shown in Figure 2.2.

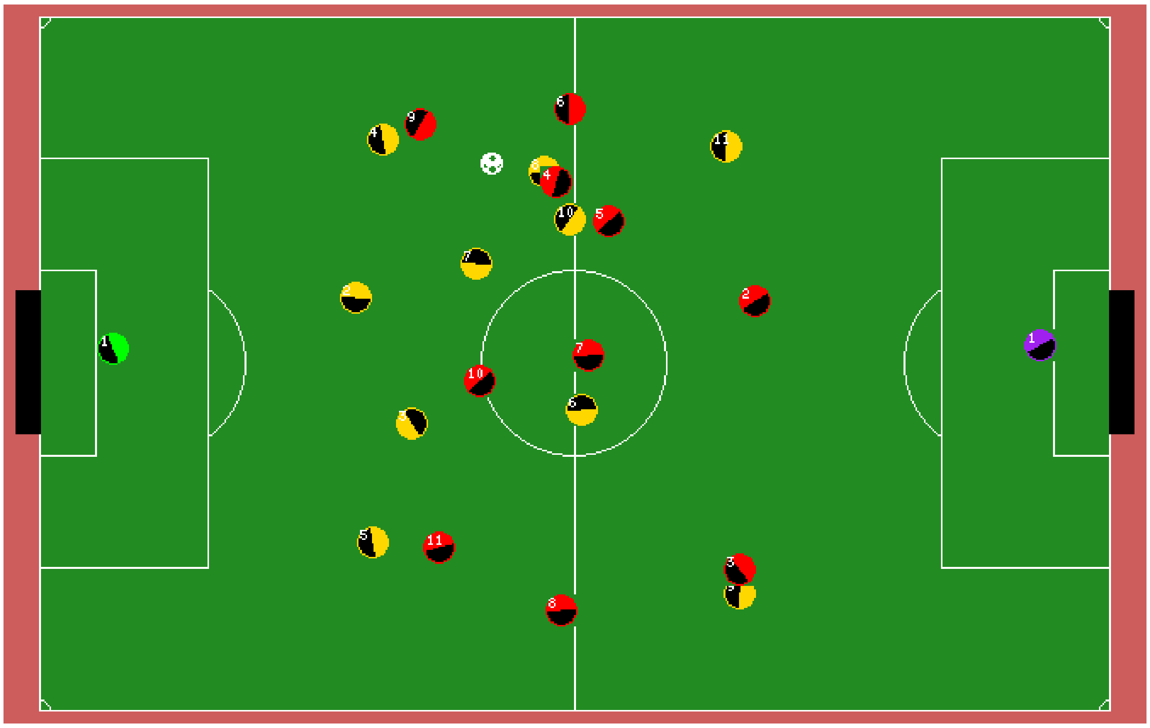


Figure 2.2: Screen shot of the Soccer Server System. The agents are represented as circles, with the light colored portion indicating which direction they are facing. Agents on the two teams are distinguished by the colors, with the light colored agents on one team and the dark colored agents on the other. The ball is the white object above and to the left of center. The coach agents are not depicted.

The agents must communicate with the server at the level of parameterized actions like turn, dash, and kick. Higher level actions, such as passing and going to a position on the field, must be implemented by combining the lower-level actions that the server understands. For example, moving to a location requires a combination of turning and dashing actions, and kicking the ball in a direction is usually a combination of a number of small kicks and player movements to accelerate the ball in the correct direction. This process, in combination with the perception and action noise of the environment, results in noisy execution of the higher level actions. In particular, there is noise in the exact result of the action and the time taken to execute it.

There are eleven independent players on each side as well as a coach agent who has a global view of the world, but whose only action is to send short messages to the players. The coach does not get information about the percepts received by the players or the actions that they have tried to execute.

Actions must be selected in real-time, with each of the agents having an opportunity to act 10 times a second. Each of these action opportunities is known as a “cycle.” Visual information is sent 6 or 7 times per second. Over a standard 10 minute game, this gives 6000 action opportunities and 4000 receipts of visual information.

The agents can communicate, but the communication is of limited bandwidth and unreliable. Further, only agents within 50m (approximately half the length of the field) of the player who is talking will be able to hear the message. The coach is able to communicate with *all* of the players on the field independent of their location. However, the coach is allowed only occasional communication with the players, namely when the play is stopped (due to an out of bounds call, kick-off, etc.) or every 30 seconds, whichever is sooner. The coach’s utterances can not be heard by the opponent.

A few additional points should be noted:

- The world can change very quickly. Based on the maximum agent speeds and size of the field, we can calculate that in about 15 seconds, the world could transition to almost any state.[1]

- The score and global clock are the only shared state features. For all other information, the agents must rely on their own local view and communication from teammates. Because of noise in sensations, the agents’ beliefs about the locations of objects in the world are inconsistent.

- There is an active and intelligent opponent team.

We used a variety of server versions throughout the thesis, from 6.xx to 9.xx. The changes between the server versions do not have a significant impact on the algorithms and techniques used here except for the standard coaching language discussed in Section 2.2.2.

[1] Agents can move at about 10m/second and the field diagonal is about 123m, plus some time to manipulate the ball if necessary.

2.2.2 CLang

In the fall of 2000, the idea of having a competition among coach agents had been around for some time in the community of RoboCup simulation league participants. While the Soccer Server had support for coach agents, more was needed to create a coach competition. The chair for the simulation league for RoboCup 2001, Gal Kaminka, pushed to make the competition happen.

In order to compare coaches, it is necessary for a coach to work with agents other than those designed by the research group which created the coach. Otherwise, it would be impossible to determine what portion of the performance of a team should be credited to the coach as opposed to the team itself. Therefore, a standard language for coaches to speak to players was needed. In the fall of 2000, the interested parties from the RoboCup simulation league community used the coach mailing list [Coach Mailing List] to collectively design the language which became known as CLang.[2] I was one of the main contributors to the language.

CLang went through a sizable update in between RoboCup 2001 and RoboCup 2002, corresponding to server versions 7.xx and 8.xx. CLang remained unchanged between 8.xx and 9.xx. The version described here is the 8.xx version. With some minor syntactic changes, almost everything in version 7.xx CLang can be written into version 8.xx CLang.

This section will cover the general structure and important aspects of CLang. This text draws from the discussion on the coach mailing list [Coach Mailing List] as well as the Soccer Server Manual [Chen et al., 2001]. The full details and grammar will not be covered, but these are provided, along with extended examples, in Appendix A for the interested reader.

The major features of CLang are:

- CLang is a rule based language. Conditions are based on various state variables, such as the ball position, and the actions are abstract actions like passing or positioning. Rules can advise the agent to do or *not* do an action.

- Regions of the field are used throughout the language. Regions can be defined by flexible geometric shapes, possibly based on the locations of mobile objects on the field (i.e. the ball and players). Regions can be additively combined. Conditions and actions both use regions in their definitions.

[2] I believe that Timo Steffens gets credit for first using the name "CLang" (standing for Coach Language), but it probably came from the fact that, when writing the code to implement the coach language in the server, I used the abbreviation "clang" to mean coach language.

- CLang has variables for player numbers. Variables can be instantiated in the condition part of the rule and then used in the action part. Unfortunately, because of the difficulty of correctly implementing the variable matching, most coachable agents support only the simple uses of variables (more details can be found in Appendix A).
- Expressions can be named. By assigning a name to an expression representing a region, condition, or other CLang element, the element can be more easily reused and changed. Names can also improve human legibility and conciseness.
- Rules can be nested. That is, the action part of a rule can refer to another rule. Nesting allows concepts to be easily embodied in a set of rules and reused.

A coach gives a number of rules to the agents, at the beginning and/or throughout the game. A rule consists of a condition (which is a logical combination of atoms about states of the world) and a directive. A directive says that particular agents on the team *should* or *should not* perform a set of actions. These actions are macro-actions like positioning and passing. In order to specify both conditions and actions, regions on the field can be defined in flexible ways.

To illustrate, here is an example of a CLang rule (note that line breaks are not significant):

```
(definerule DEFMarkPlayer direc (
   (and (not (playm bko)) (bpos "DEFRegLeft") )
   (do our {2 3 4 5} (mark {0}))))
```

A rule named "DefMarkPlayer" is defined. Its condition is that the play mode is *not* before kick off `(not (playm bko))` and the ball is in a region named "DEFRegLeft" `(bpos "DEFRegLeft")`. The definition of that region is not shown here. If the condition is true, players 2, 3, 4, and 5 are advised to mark any player on the opponents team (the set `{0}` indicates all players).

Further details can be found in Appendix A.

2.3 RCSSMaze

State and action spaces : continuous
Observability : partial
Multi-Agent/Adversarial : no

Level of advice actions : macro-actions
Optimal policy known : probably[3]
Advice frequency : one time
Limited agent(s) : Yes, but can be modeled

The last environment is designed to be part way in between the complexity of the other two. We created a maze environment we call RCSSMaze.[4] An agent (which is being coached) moves on a continuous plane. Also on the plane are agents acting as walls. Each wall is much larger than the coached agent and if the agent runs into the wall, the agent will get reset to a start state. The walls are mobile, responding to the current location of the agent and/or timed waits. All movement of the walls follows fixed finite state machines.

We implemented RCSSMaze as a modification of the Soccer Server. We intentionally left the partial, noisy observability to add noise to the maze that is not easily modeled. For example, the walls will not always realize that the agent has run into them, especially if the agent is on the edge of the wall. The agent is also not fully aware of the state of the world. The actions maintain their noise, so the agent does not always accomplish what it intends.

We require the agent to bring the ball with it as it moves through the maze. Running into a wall is then moving into the kickable area for the wall, which then kicks the ball back to the start state.

The basic interaction with the server and the simulated physics remain the same (except for the much larger size of the wall agents). With this environment, we still need to use abstract states and actions for reasoning, but it is easier to modify the functioning of the environment (notably by changing the behavior of the walls) and the reward conditions for the agent.

The agent still has limitations. We are using one particular implementation of the basic skills such as moving and looking around. The performance of these skills is imperfect. The coach can model the performance of the agent's skills by observing the agent's behavior.

One advantage of using a modification of the simulated soccer environment is that we can still use the advice language CLang as described in Section 2.2.2. While some components of the language make no sense (such as "opponents"), the remainder can still be interpreted meaningfully by the agent.

[3] We set up the mazes so that we think we know the optimal path. However, since we do not know all the transition probabilities, we can't be sure that we have the optimal path.

[4] RCSS is the prefix for the files and project in the simulated robot soccer environment. RCSSMaze stands for RoboCup Soccer Simulator Maze.

By removing some of the variables from the soccer domain, RCSSMaze is a simpler environment which allows more thorough experiments to be run. The behavior of the walls can be more precisely controlled than particular opponent teams. Further, by design of the environment, we believe that we (approximately) know the optimal policies for the environment. Therefore, this is a good test bed for the learning and use of an environment model.

We created two mazes for testing. A state of the first maze, which we call maze 0, is shown in Figure 2.3, with some additional markings for explanation. The agent starts on the left (where it is pictured) and is trying to get to the middle right hand side. While the agent is not fully aware of the state of the maze, from our global perspective, there look to be three possible routes to get there once going around the vertical line of walls. However, if the agent moves into the upper pathway, walls A and B move to block the way. If the agent moves into the lower pathway, wall C will move to block that pathway. However, if the agent goes through the middle, the wall currently blocking the way (wall D) will move and the agent can reach the reward.

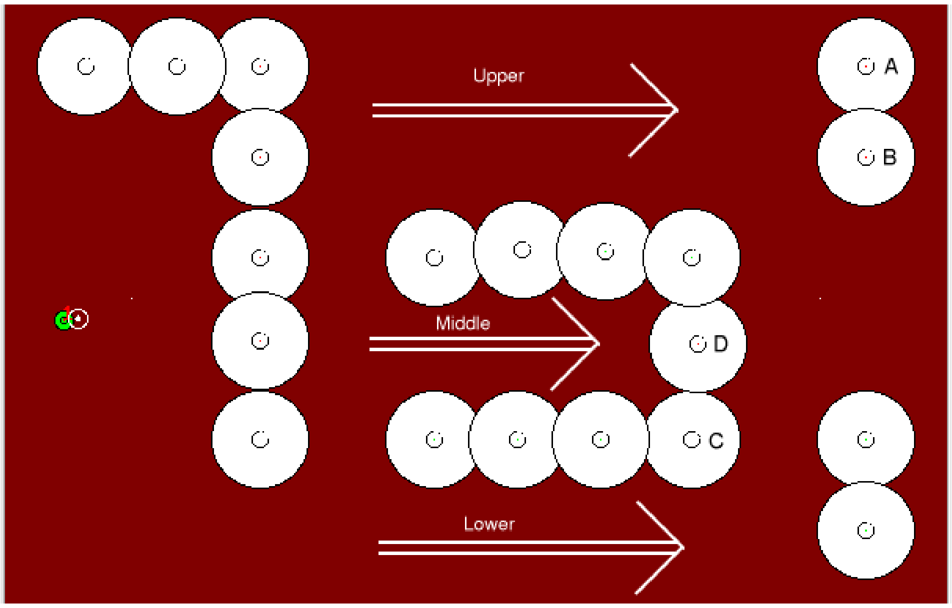


Figure 2.3: A view of maze 0. The large white circles represent walls. The small dark circle on the left with the white circle next to it is the agent to be coached.

The other maze, which we call maze 1, is a bit more complicated. The walls move on time delays as well as in response to the agent's position. A state of maze 1 is shown in Figure 2.4. There are once again apparently three possible pathways to the goal in the middle right. Walls A, B, and C in the upper pathway all move back and forth from blocking the pathway to above it every 60 to 75 cycles. The agent could get through this pathway by waiting for each wall to move, then advancing. In the middle pathway, wall D blocks and unblocks the path every 115 cycles. In addition, if the agent waits for 50 cycles next to wall D while it is blocking the path, then walls E and F will move to the upper right hand corner. This provides two ways for the agent to reach the goal through the middle pathway: either wait for wall D to move or, if that takes a while, move around wall D once walls E and F have moved away. Finally, if the agent takes the lower pathway, wall G moves to block the way.

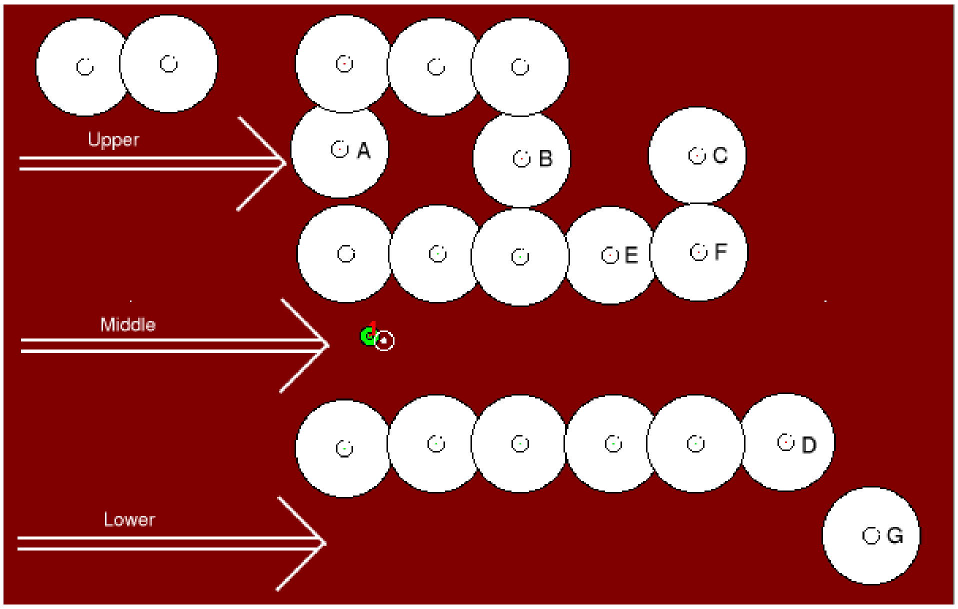


Figure 2.4: A view of maze 1. The large white circles represent walls. The small dark circle in the middle with the white circle next to it is the agent.

As should be clear, the dynamics of maze 1 are more complicated than maze 0. In addition, there are multiple possible paths to the goal and it is not clear which one is optimal.

2.4 Summary

This chapter has covered the technical details of all environments used in this thesis. The differences between them and the range of complexities was discussed. This chapter should serve as a reference when the environments are used in later chapters.

Chapter 3

Advice Giving

One of the coaching sub-questions presented in Chapter 1 was "Once the coach has desired actions for the team, how does the coach adapt advice to the agent abilities?" In this chapter, we address this question. In particular, we assume that the coach already knows how it wants the agents to act by knowing the optimal policies of the environment. However, the coach must deal with the agent's physical or mental limitations. Throughout, the agent is learning to perform in the environment at the same time that the coach is learning how best to help.

We will be using the predator-prey environment as described in Section 2.1. The coach has full knowledge of the dynamics of environment and all optimal policies. The "dynamics" include the effects of all possible actions of the agents. In order for the experiments to be as straightforward as possible, the coach's advice will be in the form of a recommended action for the predator to take. This direct form of advice allows us to focus on the learning done by the coach and agent without having to be concerned with how advice is interpreted by the agent.

All algorithms described in this chapter are fully implemented. Empirical validation is done for all algorithms under various scenarios.

3.1 Continual Advice

We first consider an agent with no restrictions on the actions it can perform and communication from coach to agent every step. The algorithms for the coach to generate advice and for the agent to incorporate advice into its behavior selection will now be explained.

3.1.1 Learning Algorithms

We begin with basic Q-learning agents. Q-learning is a reinforcement learning algorithm which does not make a model of the environment. A Q-table is maintained which, for every state and every action, maintains an estimate (based on the past rewards and transitions) of the value of taking a particular action in a particular state. The table is updated (with what I call a Q-update) every time an action is taken, based on the previous state, the action, the next state, and the reward received. At any point, a deterministic policy can be extracted from a Q-table by picking, in each state, the action which has the highest Q-value.

Q-learning is a standard machine learning algorithm, and everything described here until the addition of the coach agent is well known and well studied [e.g., Watkins and Dayan, 1992]. The state space is assumed to be represented explicitly.

During learning, the agent has a fixed exploration probability ϵ. With probability ϵ the agent takes a random action and with probability $(1 - \epsilon)$ it chooses uniformly randomly between the actions with the highest Q-value. This action selection is commonly known as ϵ-greedy exploration.

For all Q-learning done here, the learning rate decays over time. For each Q-table entry, we keep track of the number of times that entry has been updated (call it v), and calculate the learning rate for each update as $\frac{1}{1+v}$.

We modify the standard Q-learner by adding recommended actions. Every step, the coach recommends a single action. We assume that the communication is free but limited. Table 3.1 shows the algorithm used by the coached agent. The only difference with a basic Q-learner is in choosing an action. The new parameter β controls the probability that the agent will follow the coach's advice. Otherwise, the agent reverts to the normal exploration mode. Except where noted, we use $\beta = 0.9$. We use this parameter because this is advice, not control as discussed in Section 1.1, where it is suggested that advice receivers need to be able to interpret and refine advice for best effectiveness. In a real system, the decision to ignore or interpret advice will naturally be more complex than a simple random variable.

The coach is also a Q learner, but its actions are to advise *optimal* actions. The algorithm for the coach is shown in Table 3.2. While this algorithm is overly complex for this particular task, we will continue to use the same algorithm later in more complex settings. The algorithm has two primary cases (which we will experimentally vary as an independent variable): seeing the agent's action and not seeing the agent's action. If the coach sees the agent's action and the agent takes an optimal action, the coach performs a Q-update with that action. This update is based on the assumption that if the coach had

ContinualAdviceTaker(ϵ, β)
$a :=$ recommended action
with probability β
do a
with probability $1 - \beta$
Choose action (ϵ greedy)
Q-update for action performed

Table 3.1: ContinualAdviceTaker algorithm for an agent Q-learning and receiving continual advice

CoachContinualAdvice(ϵ)
$g :=$ last recommended action
if (see agent action (call it a))
if (a is optimal)
Q-update for a
else
no Q-update
else
Q-update for g
recommend action (ϵ greedy)

Table 3.2: CoachContinualAdvice algorithm for the coach Q-learning and providing advice

advised that action, the agent probably would have done it. Note that in general, there may be multiple optimal actions in a particular state. However, if the action is not optimal, the coach does nothing; the coach's Q-table does not even include non-optimal actions. If the coach does not see the agent's actions, then the coach always performs the Q-update for the recommended action. We assume that the coach does not know the algorithm (or parameters such as β) used by the advice receiver.

Note that this algorithm requires that the coach knows all optimal actions. For these experiments, the coach precomputes the optimal actions beforehand.

What the coach is learning here (and will be learning throughout) is not what the optimal actions are (the coach already knows this), but the value of the advice provided. The coach restricts its Q-table to just the optimal actions and then pessimistically initializes the table. The Q-table then provides an estimate of the value achieved by the agent when the coach *recommends* an action (which is not the same as an agent *taking* the action, notably in the case where the coach can't see the agent's actions). This updating scheme will have important consequences when the advisee agent has limitations in the following sections.

3.1.2 Experimental Results in Predator-Prey

In this section, we run experiments to establish two things. First, we need performance baselines to which to compare results in later sections. The environment setup here is the least restrictive of the ones we will consider. Second, we want to gain an understanding of how β, the amount that the agent listens to the coach, affects the speed of learning. We expect that the performance will monotonically increase with β, but we want to empirically verify this.

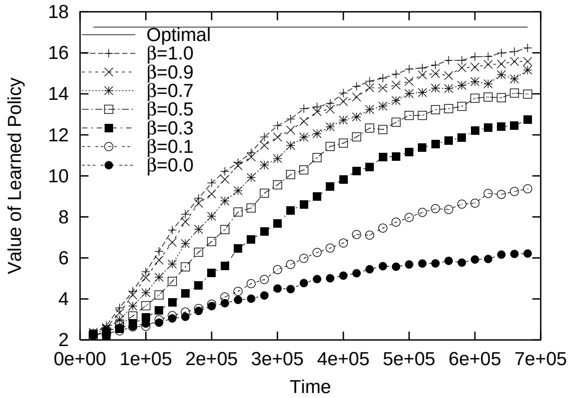
We ran the learning algorithms above in the predator-prey environment (Section 2.1) for 700,000 steps. We alternated periods of learning and evaluation. Every 5000 steps, 5000 steps of policy evaluation were done. The results throughout are the average values per step obtained during the policy evaluation periods. We ran 10 separate trials of each experimental condition and averaged the results together. In order to make the curves more readable, the values of two periods of policy evaluation were averaged together before plotting.

Throughout, we used $\epsilon = 0.1$ as the exploration parameter. The Q table was initialized to 0 everywhere, and the optimal Q values are positive everywhere. This initialization is known as pessimistic and has the effect that as soon as an action is seen to have a positive effect, it will be chosen more often.

The $\beta = 0.0$ line in Figure 3.1 shows the results for the predator agent learning without the coach. The agent is learning, though over 700,000 steps (half of which are learning, half are evaluation), the agent achieves only 36% of optimal.

The rest of Figure 3.1 presents the results of the coach advising (with the algorithm CoachContinualAdvice from Table 3.2) and the predator taking advice (with the algorithm ContinualAdviceTaker from Table 3.1), varying the value of β to the ContinualAdviceTaker algorithm. This value β controls how often the predator listens to the coach. As expected, with the coach, the predator agent learns much more quickly and reaches a nearly optimal policy. The coach is effectively guiding the agent to the right actions, improving the speed at which the predator's Q-table reflects an optimal policy. Also, the more often the predator listens to the coach (i.e. as β increases), the faster the learning and the better the performance. However, there are obviously diminishing returns as β increases.

In short, as expected, increasing the probability that the agent listens to the coach monotonically increases the performance of the agent, with diminishing returns.



Condition	% of Optimal
$\beta = 1.0$	94%
$\beta = 0.9$	90%
$\beta = 0.7$	88%
$\beta = 0.5$	81%
$\beta = 0.3$	74%
$\beta = 0.1$	54%
$\beta = 0.0$	36%

Figure 3.1: Data for a predator agent learning with a coach advising the agent every cycle. The table on the right shows the performance level achieved at the last step of learning, compared to optimal.

3.2 Limited Agent Actions

We now consider cases where the action space of the coached agent is limited. For our purposes, this will mean that the agent is not able to perform some actions. In a complex environment, differences in agent abilities can occur, and for these experiments, we will model these differences as inability to perform some actions.

3.2.1 Learning Algorithms

A revised algorithm, LimitedContinualAdviceTaker, for the coached agent is shown in Table 3.3. The only difference from ContinualAdviceTaker (Table 3.1) is that if the coach recommends an action the agent can not perform, a random action is performed. This choice is intended to simulate an agent which is not fully aware of its own limitations. By trying to do something which it can't do, some other action will result. However, we assume that for a given state and recommended disabled action, the *same* action will result every time.

The coach can follow the same CoachContinualAdvice algorithm as before (Table 3.2) and should be able to learn which optimal policy the agent can follow. If the coach can see the agent's actions, then only those Q-values for actions which the agent can perform will have their values increased. If the coach can not see the actions, then recommending an action that the agent can perform will tend to lead to a higher value state than recommend-

```
LimitedContinualAdviceTaker(ε, β)
   a := recommended action
   with probability β
      if (a is enabled)
         do a
      else
         do random enabled action
   with probability 1 − β
      Choose action with ε greedy exploration
   Q-update based on action performed
```

Table 3.3: LimitedContinualAdviceTaker algorithm for an advice receiving agent with limited actions. Note that for the random action done, the same action is chosen every time that state is visited.

ing an action the agent can not perform (since the agent then takes some other possibly non-optimal action). In this manner, the coach's Q-table should reflect an optimal policy that the agent can actually perform, and over time the coach should less often recommend an action which the predator agent can not do. This algorithm implicitly assumes that the agent can perform some optimal policy.

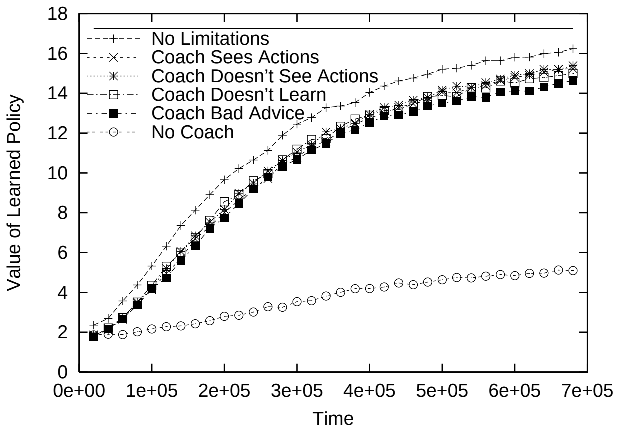
3.2.2 Experimental Results in Predator-Prey

These experiments will explore the effectiveness the algorithm CoachContinualAdvice (Table 3.2) when the receiving agent has limitations. Our hypothesis is that a learning coach will provide better advice than a non-learning coach, leading the agent to higher performance. We once again use the predator-prey world as described in Section 2.1.

We constrain the limitations of the agent such that the agent can still perform some optimal policy. In particular, in this environment, 9392 of the 46656 states have more than one optimal action. Every choice of an optimal action for each of those states gives an optimal policy. We impose a restriction on the action space of the predator such that predator agent is able to perform exactly one of the optimal policies. We once again average the results of 10 trials. Each trial has a separate set of limitations for the predator, but the various experimental conditions use the same limitations.

Further, for a given state action pair, the same random action always results when the agent is told to do an action it can not do. We also chose to use $\beta = 1.0$ to emphasize the effects of the coach's advice.

Results from this learning experiment are shown in Figure 3.2. In order to provide a reasonable point of comparison, we ran an experiment with the coach providing intentionally bad advice. That is, for every state where the predator has an optimal action disabled, the coach would always recommend that action, and the predator would always take the same random action in a given state. This experimental setup is the data line "Coach Bad Advice" in Figure 3.2. The data line for the predator learning without limitations and with the coach (with $\beta = 1.0$ from Section 3.1) is shown labeled "No Limitation."



Condition	% of Optimal
No Limitations	94%
Sees Actions	88%
Doesn't See Actions	89%
Doesn't Learn	87%
Bad Advice	85%
No Coach	30%

Figure 3.2: The value of the policy learned by a limited predator agent under various coach conditions. The table on the right shows the performance level achieved at the last step of learning, compared to optimal.

An important issue to consider when analyzing the results is how much our limitation algorithm actually limits the agent. Only 20% of the states have multiple optimal actions, and of those, almost all have exactly 2 optimal actions. If an agent performs optimally on the 80% of the state space with one optimal action and randomly anytime there is more than one optimal action, the average reward per step of this policy is approximately 14.2 (optimal is 17.3).

Whether or not the coach sees the actions, the learning coach achieves better performance than the baseline of bad advice and approaches the performance without limitations at all. The coach also performs slightly better with learning, though as the "bad advice" baseline shows, there is actually not too much room for improvement. Later experiments will show where the coach's learning can have a bigger impact.

A natural question to ask at this point is whether the coach is really learning anything about the agent's abilities. One measure of what the coach has learned about the agent is the percentage of the coach's recommended actions that the agent can not do. Figure 3.3 shows this value as the simulation progresses. The three lines represent the cases where the coach is not learning, learning and not seeing the agent's actions, and learning and seeing the agent's actions. As one would expect, with learning, the percentage of bad actions recommended goes down over time, and seeing the actions enables faster learning.

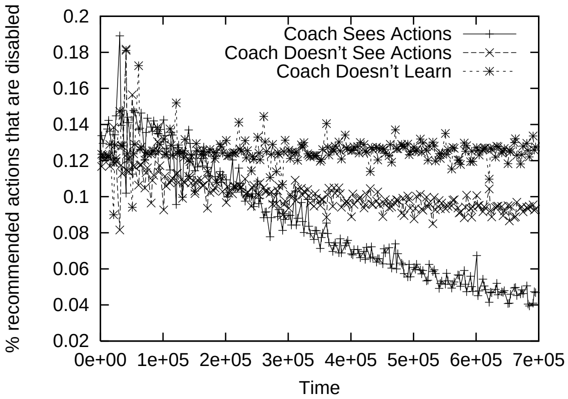


Figure 3.3: As a function of time, the percentage of coach recommended actions which the predator can not do. Each dot represents 10,000 steps.

In summary, the effect on the agent's rewards of the coach's learning while providing advice was relatively small. While this was surprising, it was demonstrated that the coach did learn about the abilities of the agent. The small magnitude of the effect on final reward appears to come from the fact that the limitations imposed on the agent do not hurt the performance a great deal. In other words, for an agent with few limitations, it is less useful for a coach to learn about the agent.

3.3 Limited Bandwidth

Up to this point, the coach was giving advice to the agent every cycle. This set of experiments deals with limiting the amount of advice provided. We still use an advisee agent with a limited action space as described in Section 3.2.

3.3.1 Learning Algorithms

To allow the coach to talk about states that are not the current state, a single piece of advice is now a state-action pair. The coach is advising an agent to perform a particular action in a particular state. We impose limitations on how much advice the coach can give. We use two parameters: I is the interval for communication and K is the number of states that the coach can advise about. Every I cycles, the coach can send K pieces of advice.

The agent stores all advice received from the coach and consults that table at each time step. The new algorithm LimitedBWAdviceTaker is shown in Table 3.4. For this experiment, the table T simply stores all advice received and if multiple actions have been advised for a given state, the table returns the first one received which the agent is capable of doing.

```
LimitedBWAdviceTaker(ε, β, T)
  if (advice received from coach)
    add advice to T
  if (current state is in T)
    a := recommended action from T
    with probability β
      do a
    with probability 1 − β
      Choose action (ε greedy)
  else
    Choose action (ε greedy)
  Q-update for action performed
```

Table 3.4: LimitedBWAdviceTaker algorithm for an advice receiving Q-learner with limited bandwidth with the coach. T is a table which stores past advice from the coach.

We propose two strategies for sending advice. The first strategy is mostly random; the coach randomly chooses K states and sends advice for an *optimal* action for each of those states (see Table 3.5). Note that while we call this a random strategy, it is still providing optimal advice for the states it chooses. If the bandwidth between the coach and advice receiver were large enough, CoachRandomState would send the entire optimal policy to the agent.

The other strategy, CoachOptQ (see Table 3.6), is more knowledge intensive. It requires the entire optimal Q table and always seeing the coached agent's actions. Like

```
CoachRandomState()
  if (time to send advice)
    do K times
      s := random state
      a := random optimal action for s
      advise (s, a)
```

Table 3.5: CoachRandomState algorithm for coach giving advice with limited bandwidth

CoachRandomState, CoachOptQ always provides optimal advice for the states it chooses. The difference is that CoachOptQ attempts to choose better states about which to advise. The basic idea is to advise about the states observed for which the agent performed the least valuable actions. Note that the *smallest* values are chosen first since all of the values in W are negative. If the algorithm runs out of states about which to advise, it simply chooses random states rather than wasting the bandwidth.

CoachOptQ learns about agent limitations in the same way that the algorithm CoachContinualAdvice (Table 3.2) does. Namely, through the Q-updates, the actions which the agent can not do will end up with lower values as discussed in Section 3.2.1.

While neither CoachRandomState nor CoachOptQ may be good algorithms to implement in a real world setting, together they provide good bounds on the range of performance that might be observed. CoachRandomState puts no intelligence into choosing what states to advise about, and CoachOptQ uses more information than would likely be available in any real world setting. The improvement that CoachOptQ achieves over CoachRandomState indicates how much benefit could be achieved by doing smarter state selection in an advice-giving framework like this one.

3.3.2 Experimental Results in Predator-Prey

In these experiments, we examine the difference in performance between the CoachRandomState (Table 3.5) and CoachOptQ (Table 3.6) algorithms. Since the CoachOptQ algorithms uses more knowledge and learns during execution, we expect its performance to be better, but we do not know to what extent or how it will vary with the bandwidth.

We once again use the predator-prey world as described in Section 3.1.2. The predator has limited actions as described in Section 3.2.2. For the parameters limiting advice bandwidth, we chose I to be 500 and K to vary between 1 and 50. Recall that every I cycles,

```
CoachOptQ(ε)
  Q* := optimal Q-table
  V* := optimal value function (V*(s) = max_a Q*(s,a))
  W := (initially empty) history of agent actions
  every cycle
    s := current state
    a := action the agent took
    put ⟨s, V*(s) − Q*(s,a)⟩ in W
    if (a was an advised action for s)
      perform Q-update
    if (time to send advice)
      W′ = (smallest K values of W)
      for each (s′, x) in W′
        if (x ≈ 0)
          s′ := random state
        a′ := action for s′ (ε greedy)
        advise (s′, a′)
        remove (s′, x) from W
      clear W
```

Table 3.6: CoachOptQ algorithm for coach giving advice with limited bandwidth.

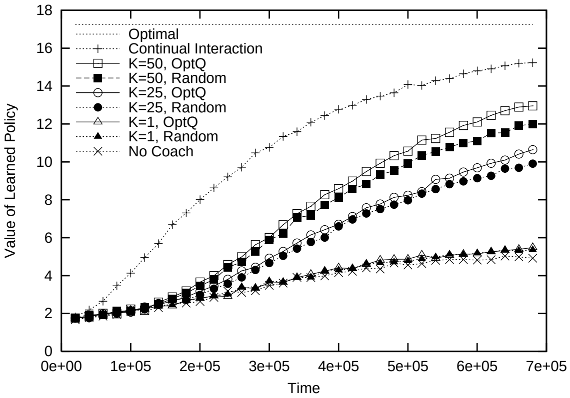
the coach can send K pieces of advice.

The results shown in Figure 3.4 are the average of 10 trials with different limitations for each trial.[1] Note first that as the amount of information the coach can send gets larger (i.e. K gets larger), the agent learns more quickly. While the same performance as the coach advising every cycle is not achieved, we do approach that performance.

Note that the $K = 1$ line is only slightly better than no coach at all. This result should not be surprising given that throughout the entire run, the coach is only able to send 1400 pieces of advice throughout the entire run, which is only 3% of the state space.

For $K = 25$ and $K = 50$, we see that CoachOptQ performs better than CoachRandomState, which was expected, but the difference is not dramatic. We identify two factors which contribute to the smaller than expected difference. First, by the end of the simula-

[1]In this case, different limitations were used for the different values of K. Since we are averaging over 10 trials, any effect of a more or less difficult predator limitation should average out.



Condition	% Optimal
Continual Interaction	88%
$K = 50$, CoachOptQ	75%
$K = 50$, CoachRandomState	70%
$K = 25$, CoachOptQ	62%
$K = 25$, CoachRandomState	57%
$K = 1$, CoachOptQ	32%
$K = 1$, CoachRandomState	31%
No Coach	29%

Figure 3.4: Policy evaluation for the predator for various coach strategies and values of K. Note that open symbols are always for the CoachOptQ strategy and the filled in symbol of the same type is for CoachRandomState at the same K level. Note that the $K = 1$ lines overlap and are therefore not both visible. The table on the right shows the percent of optimal for the policy reached at the final step.

tion, the CoachRandomState strategy achieves fairly large coverage of the state space. For example, with $K = 25$, the random strategy is expected to provide advice for about 47% of the states.[2] Since the agent remembers all of the advice, near the end of the run it is essentially getting optimal advice for half the states. Secondly, the CoachOptQ strategy only considers states in which the agent has already been and taken an action before reverting to the same behavior as CoachRandomState. The chance of encountering one of these states again is about the same as encountering any other given state.

In summary, while CoachOptQ always performed better than CoachRandomState, the differences were surprisingly not dramatic. The next section continues to explore conditions which affect the relative performance of CoachRandomState and CoachOptQ.

3.4 Limited Bandwidth and Memory

In Section 3.3, the coached agent had a limited action space and there was limited bandwidth for communication between the agents. However, the coached agent remembered all

[2] With basic probability theory, one can calculate that if T is the number of pieces of advice and S the size of the state space, the expected number of distinct states about which the random strategy advises is $S(1 - (\frac{S-1}{S})^T)$. At the end of 700,000 steps with $K = 25$, 35,000 pieces of advice have been given. With $S = 46,656$, the expected number is approximately 22,035 states.

advice which the coach has sent. We next explore the additional limitation of varying the amount of advice which can be remembered. The amount of advice may be an important consideration for learners that suffer from the utility problem [Minton, 1988].

3.4.1 Learning Algorithms

We consider a straightforward FIFO (First-In, First-Out) model of memory. The coached agent has a fixed memory size, and when it is full, the agent forgets the oldest advice.

The coach strategies are the same as before: CoachRandomState (Table 3.5) and CoachOptQ (Table 3.6). The coached agent still uses LimitedBWAdviceTaker (Table 3.4), but now the state advice table T only stores the last M pieces of advice heard, where M is an independent variable which we vary.

3.4.2 Experimental Results in Predator-Prey

These experiments explore how the addition of limited bandwidth affects the relative performance of the CoachRandomState (Table 3.5) and CoachOptQ (Table 3.6) algorithms for the coach. We hypothesize that in this more restricted environment, the difference in performance will be greater.

We once again use the predator-prey world as described in Section 3.1.2. The predator has limited actions as described in Section 3.2.2. Figure 3.5 and Table 3.7 show the results. With the smallest memory of $M = 1000$, the predator does not improve much over having no coach at all. This result can be explained simply because the memory can only hold approximately 2% of the state space. For that memory level, note that the most improvement (8%) is achieved by using CoachOptQ (for both $K = 25$ and $K = 50$). As the amount of memory is increased, the agent's performance improves. For $K = 1$, throughout the entire simulation the coach only sends 1400 pieces of advice, just barely more than the smallest memory. Therefore, we can not expect increasing the memory to improve performance for $K = 1$ since memory is not the limiting resource for most of the simulation.

For the other cases, we see the same general effects of the differences between the CoachRandomState and CoachOptQ strategies here as when there was no limited memory (Section 3.3.2). The limited memory size effectively lowers the absolute performance of all conditions. However, the difference between the two strategies shows that benefit can be once again be obtained if the coach learns about the capabilities of the agent.

Overall, the CoachOptQ algorithm always performs better than CoachRandomState.

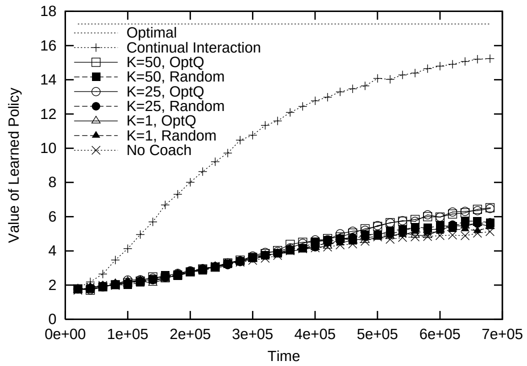
	% of Optimal			
Condition	$M = 1000$	$M = 5000$	$M = 10000$	$M = 15000$
$K = 1$, CoachOptQ	31%	30%	31%	32%
$K = 1$, CoachRandomState	31%	32%	31%	32%
$K = 25$, CoachOptQ	38%	49%	56%	59%
$K = 25$, CoachRandomState	33%	43%	50%	54%
$K = 50$, CoachOptQ	38%	54%	62%	66%
$K = 50$, CoachRandomState	33%	44%	54%	59%
No Coach	30%			

Table 3.7: The percent of optimal for the policy reached at the final step of learning for various experimental conditions. K is the number of pieces of state-action advice that can be sent every 500 steps and M is the size of the memory of the agent. See Figure 3.5 for the graphs of the policy learning over time.

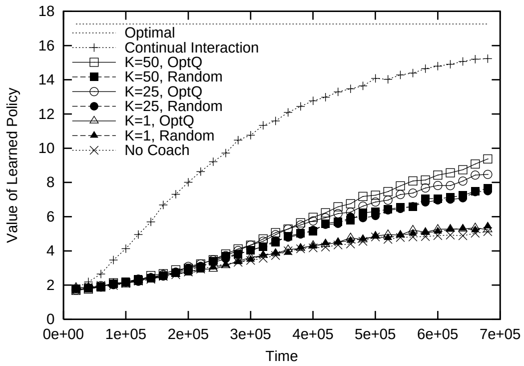
Further, the difference in performance is greater in this more limited setting of limited memory than in the previous set of experiments (see Section 3.3.2). These results suggest that the more limitations faced by the coach/agent, the more useful it is for the coach to learn about the abilities and current knowledge state of the coached agent.

3.5 Summary

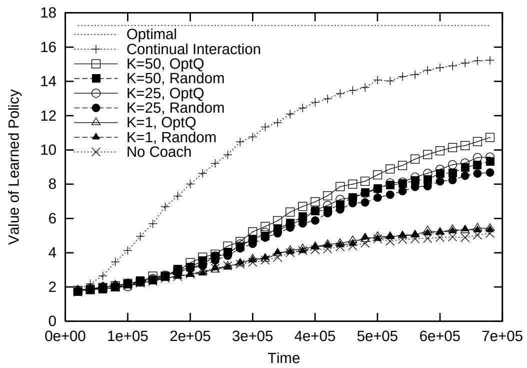
This chapter has covered a number of algorithms for a learning coach to advise actions to a learning agent. Variations in the abilities and memory of the advisee agent as well as variations in the bandwidth between coach and advisee were explored. Collectively, the results in the predator-prey domain suggest that intelligent learning by the coach of the agent's limitations can lead to improvement, though dramatic improvement was not seen. Further, higher bandwidth and memory give a greater opportunity for coach learning, though at the extreme of constant advising interaction, the importance of the coach learning diminishes. However, the more limited the setting (in bandwidth, memory, and agent abilities), the greater (relative) effect the learning of the coach can have on the performance of the agent.



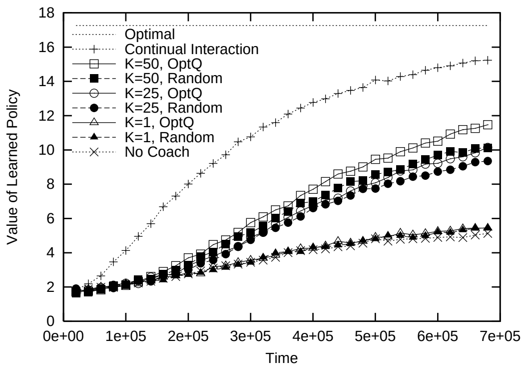
$M = 1000$; 2% of state space



$M = 5000$; 11% of state space



$M = 10000$; 21% of state space



$M = 15000$; 32% of state space

Figure 3.5: Policy evaluation during learning with a limited predator, limited bandwidth, and limited memory. M is the number of pieces of advice the predator can remember. K is the number of pieces of advice the coach can say at each interval of 500 cycles. Note that open symbols are always for the CoachOptQ strategy and the filled in symbol of the same type is for CoachRandomState at the same K level. See Table 3.7 for the percent of optimal reached for each condition.

Chapter 4

Learning Advice

Chapter 3 discussed the problems faced by a coach in giving good advice once the coach knows the optimal policies for the agent(s) in the environment. In this chapter, we now address the question of where this understanding of the environment can come from. The coach passively observes the environment and then, based on those observations, constructs an abstract model of the environment. This model can then be used to generate advice for the agents.

In this chapter we present:

- A set of algorithms to learn an abstract Markov Decision Process model of an environment from observations (Section 4.2)
- The representations and details of our abstract state spaces (Section 4.3)
- Experimental results in RCSSMaze and several variations of simulated soccer (Section 4.4)

4.1 Overview

Markov Decision Processes (MDPs) are a well-studied formalism for modeling an environment and finding a good action policy. The use of both state and action abstraction has received significant attention (see Section 7.5). In this research, we introduce algorithms to construct an MDP from observed agent executions. The MDP uses both state abstraction and temporally extended abstract actions.

The route from observations or logs of games to a useful model for advice giving has a number of steps. Figure 4.1 gives an overview of the entire process. The formalisms mentioned will be explained throughout this section.

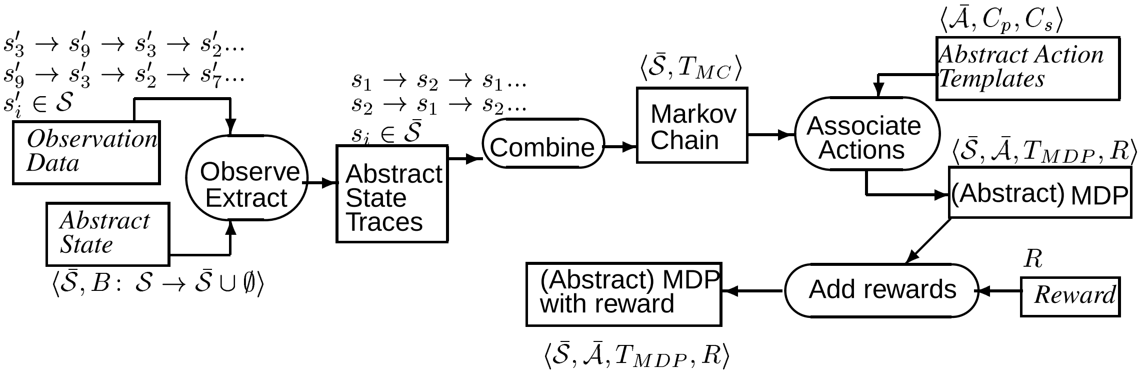


Figure 4.1: The process of a coach learning a Markov Decision Process from observation. The symbols are examples or types based on the formalisms presented. The boxes with italics indicate information that must be provided to the system.

We now overview our algorithms for creating a Markov Decision Process from observations. Our goal is to learn an *abstract* MDP. The main steps of the process are:

- Transform observations into sequences of abstract states. Note that the observations are series of states, without any information about actions performed.
- Create a Markov Chain based on the observed transition frequencies among the abstract states.
- Transform the Markov Chain to a Markov Decision Process (MDP) by associating transitions to actions based on abstract action templates. The action specifications notably do not include any probability information; all probabilities are estimated from the observed data.
- Add rewards to the MDP.

We then use this abstract MDP to provide advice to agent(s). However, while we use the learned MDP for producing advice, an environment model can be useful in other contexts and our learning algorithms are not limited to advice production uses.

4.2 Model Learning Algorithms

This section describes in detail the algorithms to go from a set of observed executions in the environment to an abstract Markov Decision Process and then to advice.

4.2.1 Observations to Markov Chain

There are two initial inputs to the process:

Observation data consisting of sequences of observed states. These sequences can come from recordings of past performance or from online observation. Let $\mathcal{S}$ be the observed state space. The observation data is then a list of sequences of $\mathcal{S}$, or in other words, a list of $\mathcal{S}^*$. Note that we will be learning a fully observable MDP, so the observational state space should capture enough of the underlying environment to adequately capture the dynamics of the system.

State abstraction consisting of an abstract state space $\bar{\mathcal{S}}$, and an abstraction function $B\colon \mathcal{S} \to \bar{\mathcal{S}} \cup \varepsilon$. The symbol ε represents a null value. If B maps an element of $\mathcal{S}$ to ε, this indicates there is no corresponding abstract state.

Observe and extract as shown in Figure 4.1 can then be implemented in terms of the above. B is applied to every element of every sequence in the observation data. Any elements that map to ε are removed. *Observe and extract* outputs *abstract state traces* as a list of $\bar{\mathcal{S}}^*$.

Given the state traces, the *combine* algorithm produces a Markov Chain, a tuple $\langle \bar{\mathcal{S}}, T_{MC} \rangle$ where $\bar{\mathcal{S}}$ is the set of abstract states and $T_{MC}\colon \bar{\mathcal{S}} \times \bar{\mathcal{S}} \to \mathbb{R}$ is the transition function. $T_{MC}(s_1, s_2)$ gives the probability of transitioning from s_1 to s_2. Since the state space $\bar{\mathcal{S}}$ for the Markov Chain has already been given, all that is left is to calculate the transition function T_{MC}.

The *combine* algorithm estimates the transition probabilities based on the observed transitions. The algorithm calculates, for every pair of states $s_1, s_2 \in \bar{\mathcal{S}}$, a value c_{s_1,s_2} which is the number of times a transition from s_1 to s_2 was observed. The transition function is then, $\forall s_1, s_2 \in \bar{\mathcal{S}}$:

$$T_{MC}(s_1, s_2) = \frac{c_{s_1,s_2}}{\sum_{s \in \bar{\mathcal{S}}} c_{s_1,s}} \tag{4.1}$$

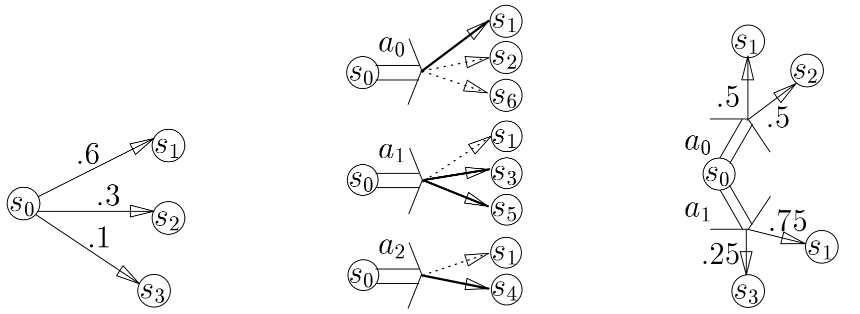
4.2.2 Markov Chain to MDP

The next step in the process is to convert the Markov Chain into a Markov Decision Process. A Markov Decision Process (MDP) is a tuple $\langle \bar{\mathcal{S}}, \bar{\mathcal{A}}, T_{MDP}, R \rangle$. $\bar{\mathcal{S}}$ is the set of abstract states, $\bar{\mathcal{A}}$ is the set of (abstract) actions, $T_{MDP}\colon \bar{\mathcal{S}} \times \bar{\mathcal{A}} \times \bar{\mathcal{S}} \rightarrow \mathbb{R}$ is the transition function, and $R\colon \bar{\mathcal{S}} \rightarrow \mathbb{R}$ is the reward function. Similar to a Markov Chain, the transition function $T_{MDP}(s_1, a, s_2)$ gives the probability of transitioning from s_1 to s_2 given that action a was taken.

In our approach, a set of abstract action templates must be specified as two sets of transitions between abstract states. A key feature of our representation is the division of the transitions into *primary* and *secondary* transitions. Primary transitions are intended to represent the normal or intended effects of actions. Whenever a primary transition of an action is observed, it is assumed that it is possible to take that action. Secondary transitions represent other possible effects of the actions. In many cases, these secondary transitions will represent a failure of the intended action. Jensen et al. [2004] use this same sort of definition to generate fault-tolerant plans.

We believe that specifying just the possible transitions associated with an action will be much easier for a domain expert than trying to specify the full transition probabilities. As will be discussed below, the separation of primary and secondary actions allows our algorithms to better identify what actions are possible at a given state.

We initially assign a zero reward function. Describing the algorithms for associating actions and transitions takes up the bulk of this section. We first illustrate the process using the example shown in Figure 4.2, then provide the full details.



(a) Markov Chain State (b) Action Templates (c) MDP State

Figure 4.2: Associating actions with the transitions from one state in a Markov Chain to create a Markov Decision Process state. In (b), the solid arrows are primary transitions and the dotted arrows are secondary transitions.

Our algorithm processes each abstract state in turn. Figure 4.2(a) shows the state s_0 in the Markov Chain. The algorithm must determine which actions can be applied in each state and how to assign the transitions and their probabilities to actions.

The transitions for the actions a_0, a_1, and a_2 are shown in Figure 4.2(b), with the primary transitions in bold. An action is added for a state if any of the action's primary transitions exist for this state. In Figure 4.2(c), actions a_0 and a_1 have been added, but a_2 has not since its primary transition $s_0 \to s_4$ was not in the Markov Chain. Primary transitions are intended to represent normal or intended effects of an action. Therefore, this step of the algorithm only attaches an action to a state if such a normal or intended transition is observed.

Once an action has been added, all transitions in the Markov Chain which are primary or secondary transitions for the action are assigned to the action. Primary or secondary transitions that are part of the action definition are not *required* to be in the Markov Chain (e.g. the $s_0 \to s_6$ transition for a_0 and the $s_0 \to s_5$ for a_1). Once all actions have been processed, the probability mass for each transition is divided equally among all repetitions of the transition and the resulting distributions are normalized. For example, in Figure 4.2, the probability mass of 0.6 assigned to the $s_0 \to s_1$ transition is divided by 2 for the 2 repetitions. The transitions $s_0 \to s_1$ and $s_0 \to s_3$ have, respectively, probabilities 0.3 and 0.1 before normalization and 0.75 and 0.25 after normalization.

Why is the original probability mass is divided equally among the repetitions? Some set of actions were actually taken by agents in order to produce the observed transitions. We do *not* have information about the distribution of actions that produced these transitions. Therefore, for a given transition, we assume that the transition was caused equally often by each of the actions. The effect of this assumption can be seen in the experiments in Section 4.4.3. This is one area where future work could improve the algorithm.

Formally, an abstract action set $\bar{\mathcal{A}}$ and functions giving the primary ($C_p\colon\ \bar{\mathcal{A}} \to \mathcal{P}(\bar{\mathcal{S}} \times \bar{\mathcal{S}})$) and secondary ($C_s\colon\ \bar{\mathcal{A}} \to \mathcal{P}(\bar{\mathcal{S}} \times \bar{\mathcal{S}})$) transitions must be given. We will calculate an unnormalized transition function T'_{MDP} which will be normalized to T_{MDP}. The complete algorithm for adding actions to the Markov Chain is shown in Table 4.1.

As noted, a null action can be added. This occurs if a transition from the Markov chain is not part of any of the actions created for this state (i.e. if $c_{s_i} = 0$ and $T_{MC}(s, s_i) \neq 0$). If everything about the model were perfect, this would not be needed. That is, if the primary and secondary transitions were complete and correct for all actions and the state abstraction maintained all relevant information about the dynamics of the world, then the null action would not be created. However, it is difficult if not impossible to be perfect in this respect. As long as not too many null actions are created, the learned MDP should

For all $s \in \bar{\mathcal{S}}$
  Let $\mathcal{T} \subseteq \mathcal{P}(\bar{\mathcal{S}} \times \bar{\mathcal{S}})$ be the transitions from s
    $\mathcal{T} = \left\{ \langle s, s_i \rangle \mid s_i \in \bar{\mathcal{S}}, T_{MC}(s, s_i) > 0 \right\}$
  Let $\mathcal{N} \subseteq \bar{\mathcal{A}}$ be actions with primary transitions for s
    $\mathcal{N} = \left\{ a \in \bar{\mathcal{A}} \mid C_p(a) \cap \mathcal{T} \neq \emptyset \right\}$
  For all $s_i \in \bar{\mathcal{S}}$
    Let c_{s_i} be the number of actions for $s \to s_i$
      $c_{s_i} = |\{ a \in \mathcal{N} \mid \langle s, s_i \rangle \in C_p(a) \cup C_s(a) \}|$
  For all $a \in \mathcal{N}$
    For all $\langle s, s_i \rangle \in C_p(a) \cup C_s(a)$
      $T'_{MDP}(s, a, s_i) = \dfrac{T_{MC}(s, s_i)}{c_{s_i}}$
  Add null action if needed (see text)
Normalize T'_{MDP} to T_{MDP}

Table 4.1: Associating actions to convert a Markov Chain to an MDP.

still be useful. In order to catch errors in action definitions, the transitions which can be assigned to the null action can be restricted, and if a disallowed one occurs, an error in the action definition can be indicated.

The definition of primary and secondary transitions and the algorithm above support situations in which the possible effects of an abstract action are known, but the probabilities of occurrence of the various results are not.

One assumption about the process should be restated. For a given transition in the Markov Chain, if several actions have that transition in their templates, then the probability mass is divided equally among the actions. If the distribution of actions which generated the observations is skewed or if the actual transition probabilities are greatly different, this algorithm can misestimate the probabilities.

Also, note that this algorithm can not properly learn the transition probabilities for two actions whose effects differ only in their probabilities. For example, consider a maze environment with just left and right actions where the actual transition probabilities are 90% for the direction intended and 10% for the other way. However, for any set of observations, the algorithm here would equally divide the probability mass among the two actions, making them appear identical.

Once we have a Markov Decision Process, any reward function can be applied. By changing the reward function, the learned Markov process can be used to produce different behaviors. This is explored further in the empirical section below.

4.2.3 MDP to Advice

Finally, we go from a Markov Decision Process to advice. The first step is to solve the MDP. Since we have all transition probabilities, we use a dynamic programming approach [Puterman, 1994]. This algorithm gives us a Q-table, where for all $s \in \bar{\mathcal{S}}$ and $a \in \bar{\mathcal{A}}$, $Q(s, a)$ gives the expected future discounted reward of taking action a in state s and performing optimally afterwords. An optimal policy (a mapping from $\bar{\mathcal{S}}$ to $\bar{\mathcal{A}}$) can be extracted from the Q table by taking the action with the highest Q value for a given state.

States and actions for advice must then be chosen. Advising about all states can overload the communication between the coach and agents (which may be limited) and stress the computational resources of the agent applying the advice [Minton, 1988]. Therefore, the scope of advice is restricted as follows:

- Remove states which don't have a minimum number of actions. For states with many possible actions, the agents will in general be in more need of advice from the coach. We experimented with different values, but for the experiments here, we only removed states without any actions.
- Remove states whose optimal actions can't be translated into the advice language.
- Only advise actions which are close to optimal. We only want to advise "good" actions, but we must be specific about what good means. We only advise actions whose values are within a given percentage of the optimal action for the given state in the learned model.

Once the states and actions which to advise have been determined, they must be translated into the advice language.

4.2.4 Advice in CLang

CLang (Section 2.2.2) is the advice language for both the simulated robot soccer environment (Section 2.2) and RCSSMaze (Section 2.3). In converting the solved MDP to CLang advice, we pruned any action dealing with actions that the opponent takes. Clearly, we can not advise the agents to perform these actions.

The coach will be sending all the advice for all states at one time. That is, before the agent has executed anything, the coach will send a set of rules providing advice for many states which may be observed in the future..

After pruning, the coach is left with a set of pairs of abstract states and abstract actions. The goal is to structure the advice such that it can be matched and applied quickly at run time by the agents. We use the structured abstract state representation discussed in Section 4.3.1 to construct a tree of CLang rules. At each internal node, one factor from the state representation is matched. At each leaf, a set of actions (for the abstract state matched along the path the the leaf) is advised. The example in Section A.2.3 is part of such an advice tree.

4.3 Abstract State and Action Spaces

This section covers in more detail our representation for abstract states and actions. Section 4.3.1 covers the tree-based representation we use for combining abstract state factors into an abstract state space. Section 4.3.2 describes the abstract state factors used for the soccer and RCSSMaze environments. The exact state space used differed for the various experiments in Section 4.4, so we leave the explanation of the trees which combine the factors until that section. Lastly, Section 4.3.3 discusses how the primary and secondary transition functions were constructed, both the general principles and some of the details for the soccer and RCSSMaze environments.

4.3.1 Tree Factored Representation

Factored state representations are a popular way to reason about state spaces for MDPs (see Section 7.5). Typically, this means describing the state space as the conjunction of a number of state factors rather than with a simple state index. For example, I could describe the world by “It is *not* raining,” “I *do* have an umbrella,” and “I am at school” rather than “I am in state 17 of 32.” In addition to being more convenient for some purposes, use of a factored representation can result in significant computational speedups.

We use factored abstract state spaces for all abstractions in this work. Because we reason with explicitly represented Q-tables and MDPs, we want the size of the resulting state space to be as small as possible. Therefore, we use a tree based representation that combines factors with both the standard AND combination and an OR combination. This section describes the representation and manipulation algorithms.

Part of the purpose of the abstract state representation is to map the observation space into the abstract state space. As discussed in Section 4.2.1, the abstraction function B can return the null value ε to indicate that no abstract state corresponds to this observed state. We allow every factor to return this null value to indicate that the factor is not applicable to the observed state.

The combination operations will map values of each factor into a natural number, with a range of the size of the combined state space. First, we need a few notes on notation. For an abstract state factor F, let B_F be the abstraction function, mapping the observed state space to an integer index. Note that B_F can return the null value ε. Let $|F|$ represent the size of the range of the output of B_F (not including the null value) i.e. the size of the factor.

The standard combination of state factors is an AND combination. For state factors F and G and for an observed world state w:

$$B_{F \text{ AND } G}(w) = \begin{cases} \varepsilon & B_F(w) = \varepsilon \\ \varepsilon & B_G(w) = \varepsilon \\ B_F(w)|G| + B_G(w) & \text{o.w.} \end{cases} \tag{4.2}$$

With this definition, $|F \text{ AND } G| = |F||G|$. The multiplication of the state space size can quickly lead to unmanageably large spaces. We therefore use an OR combination as well. First, note that the OR is ordered (i.e. it is not commutative) though it is still associative. Informally, the value of F OR G comes from the first factor which does not return ε. Formally, for an observed world state w:

$$B_{F \text{ OR } G}(w) = \begin{cases} B_F(w) & B_F(w) \neq \varepsilon \\ B_G(w) & |F| + B_F(w) = \varepsilon \end{cases} \tag{4.3}$$

With this definition, $|F \text{ OR } G| = |F| + |G|$. An OR can be used in cases where some values of the first factor make the value of the second factor irrelevant.

Once we have the basic AND and OR combination operations, a tree can be used to represent arbitrary combinations of factors. All the internal nodes must be an AND or OR and every leaf must be a state factor.

The use of OR factor combination allows different factors to be easily used for describing different parts of the state space. For example, in the state spaces used in the soccer environment, different factors are used when the ball is in play and when the ball is out of bounds.

One other part of our state abstraction representation needs to be discussed. We also allow "filters" to be attached to any AND or OR. A filter is just a predicate on the world, i.e. a function from world state to true/false. If the filter returns false, the the factor returns ε no matter what the value of the children.

4.3.2 Soccer State Spaces

We use similar spaces for both the RCSSMaze and simulated soccer environments. In this section we describe the abstract state factors and filters that we use. The exact combinations (in the tree structure described in Section 4.3.1) will be given in the experimental sections (see Section 4.4).

The abstract state factors are:

$$\textbf{Goal} \begin{cases} 0 & \text{we scored} \\ 1 & \text{they scored} \\ \varepsilon & \text{o.w.} \end{cases}$$

$$\textbf{Dead Ball} \begin{cases} 0 & \text{our free kick} \\ 1 & \text{their free kick} \\ \varepsilon & \text{o.w.} \end{cases}$$

Ball Grid Location of the ball on a discretized grid for the field (see Figure 4.3).

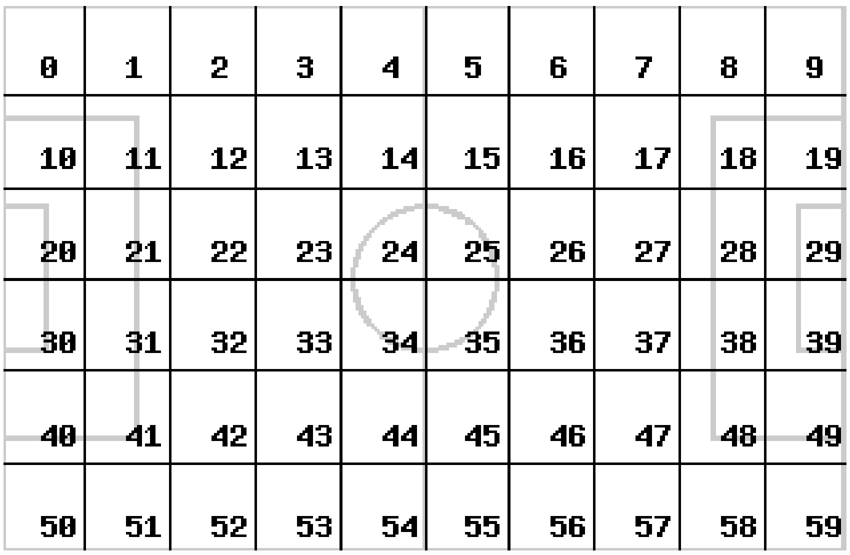


Figure 4.3: The regions for the Ball Grid state factor

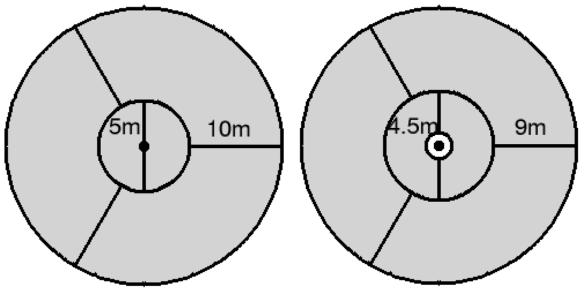


Figure 4.4: "Ball Arc" player occupancy regions. The picture on the left shows the regions for the opponent players and the picture on the right shows the regions for our teammates. As shown by the shading, the innermost circle on the right is not a region to be considered. In both cases the ball is at the center of the regions and there are 5 regions.

Ball Owner $\begin{cases} 0 & \text{we own the ball} \\ 1 & \text{they own the ball} \\ 2 & \text{neither team owns the ball} \end{cases}$

The ball owner is defined as the last agent to touch the ball. If multiple agents touched it last or the game is in a neutral restart, no one owns the ball.

Player Occupancy Presence of teammate and opponent players in defined regions. We use several different region sets for the various experiments (note that in the figures the regions are centered around the ball and oriented so that the right is always towards the attacking goal):

Ball Arc The regions are shown in Figure 4.4. We use different regions for the opponent agents and our agents. In each case, each region has two values, whether there is or is not a player of the correct type in the region.

Ball Path The regions are shown in Figure 4.5. Only one set of regions is used for both the teammates and the opponents. Each region still has two possible values but now the value is 1 only if there are *more* teammates than opponents in the region.

Ball Grid The regions are shown in Figure 4.6. These regions will be used for the RCSSMaze environment, so there is no teammate/opponent distinction. The regions measure whether any wall agent is in the region.

Note that there are varying number of regions; "Ball Arc" has 10, "Ball Path" has 5, and "Ball Grid" has 4. This means that the state space size resulting from using these different region sets will vary.

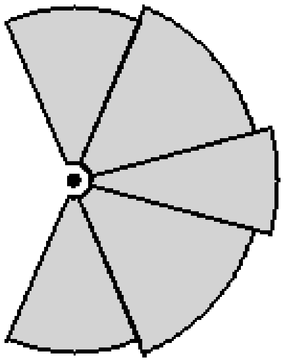


Figure 4.5: "Ball Path" player occupancy regions.

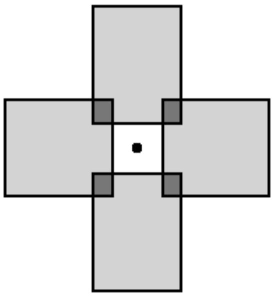


Figure 4.6: Ball Grid player occupancy region. There are four regions that overlap, as shown by the areas in darker gray. The ball is in the middle of the regions.

Maze Ball Kickable $\begin{cases} 0 & \text{the agent can kick the ball} \\ 1 & \text{a wall can kick the ball} \\ 2 & \text{both the agent and a wall can kick the ball} \end{cases}$

This factor is specifically for the RCSSMaze environment.

In addition, we use a number of filters as described in Section 4.3.1. Filters are functions from the observed state space to true or false. The filters we use are:

Play Mode True if the play mode equals a particular value. The play mode represents the state of the game, such as kick off left, goal kick right, etc. "Play on" is the mode for normal play time.

Ball Kickable True if the ball is kickable for some agent.

Ball at Right Center True if the ball is in a 5.5m x 18m region at the far right center of the area. In the soccer environment, this corresponds to one of the goal boxes.

Ball at Left Center True if the ball is in a 5.5m x 18m region at the far left center of the area. In the soccer environment, this corresponds to one of the goal boxes.

4.3.3 Transition Classification for Actions

The crucial step in converting a Markov Chain to a Markov Decision Process involves the use of action templates to identify the possible actions and assign the observed transition probabilities to the correct actions.

For the soccer domain, the abstract action space $\bar{A}$ was constrained by the advice actions in CLang. CLang supports abstract actions like passing, dribbling, and clearing (kicking the ball away). All these actions can take a parameter of an arbitrary region of the field. Actions should correspond to changes in the abstract state space. Since many CLang actions involve ball movement, we chose to consider the ball movement actions with regions from the discrete grid of ball locations (see Figure 4.3) as the parameters.

We use several hand-coded steps to construct the C_p and C_s functions describing the primary and secondary transitions for the actions.

Transitions are pairs of abstract states. We first identify a set of useful features about pairs of abstract states. For example, we have features such as "we had the ball," "is their free kick," and "ball moved nearby." All of these features are defined solely in terms of the values of the abstract state factors making up each abstract state (see Section 4.3.2 for details about the factors). Every feature is a boolean value and any number of features can be true for a given abstract state pair.

Based on these features, we then create classes of transitions. These classes summarize the difference between the pair of abstract states. For example we have classes for "our kick out of bounds, "our long pass or clear," and "our hold ball." Several points to note are:

- There can be ambiguity in what a transition means. For example, in addition to the "our long pass or clear" class, we have one for "our dribble or short pass." Since the abstract state does not record *which* agent controlled the ball, dribbling and short passes can result in the same abstract state sequence. Therefore, classes do not necessarily represent just one kind of action or event.
- The classes are closely associated with actions, but are not identical to them. Rather, each class more closely represents a possible effect from an action.
- Each transition is assigned to exactly one class. In particular, our classes are ordered and the first matching class is used.

Finally, the C_p and C_s functions describing the primary and secondary transitions are defined by giving sets of classes for the primary and secondary transitions for each action.

Appendix B describes the features, classes, and abstract actions used for MDP learning for both the soccer and RCSSMaze environments.

4.4 Experimental Results

The MDP learning and advice generation is fully implemented in the simulated soccer (Section 2.2) and RCSSMaze (Section 2.3) environments. A version of the system described here made up the bulk of the Owl entry to the RoboCup 2003 and 2004 coach competitions. This section describes our empirical validation in a number of scenarios. We begin with the soccer environment, starting with a circle passing task that is simpler than the whole game. We then move on to experiments on the entire soccer game with a variety of opponents. Finally, we present results in the RCSSMaze environment, including an additional exploration of the effect of varying the input data for learning.

Throughout this section, we will be using a variety of agents, some which are coachable (meaning that they understand the CLang advice language) and some which are not. Table 4.2 lists the agents used. The nickname will be used for the remainder of the section.

Nickname	Full Name	Year	Coachable?	Institution
UTA	UTAustinVilla	2003	Yes	Univ. Texas at Austin
CM3	Wyverns	2003	Yes	Carnegie Mellon
CM4	Wyverns	2004	Yes	Carnegie Mellon
SM	SIRIM FC	2003	No	SIRIM Berhad, Malaysia
EKA	EKA - PWR	2003	No	Wroclaw Univ. of Technology, Poland
BH	Bold Hearts	2003	No	Univ. of Hertfordshire, UK
UVA	UvA Trilearn	2003	No	Univ. of Amsterdam, Netherlands

Table 4.2: List of agents used in the experiments. The year specifies which RoboCup the team participated in (or which version for teams that competed for multiple years).

4.4.1 Soccer: Circle Passing

We constructed a sub-game of soccer in order to clearly evaluate the MDP learning from observed execution and the effect of automatically generated advice. These experiments will serve to verify that the model learned via the algorithms above is sufficient to provide advice to the agents. Additionally, the application of several different reward functions is demonstrated.

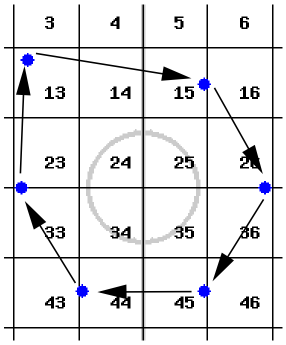


Figure 4.7: Locations and directions of passing for circle passing (cf. Figure 4.3).

We set up agents in a circle around the middle of the field and wrote a set of advice rules which cause the agents to pass in a circle, as shown in Figure 4.7.[1] The goal was to use this data to learn a model which could then be used to generate advice for a different team. This team would not know of the performance of the original team or even what the reward states in the model are. The goal was *not* to replicate this passing pattern, but to achieve a specified reward.

We ran 90 games of the UTA players executing this passing pattern. Note that because of noise in the environment, not every pass happens as intended. Agents sometimes fail to receive a pass and have to chase it outside of their normal positions. Kicks can be missed such that the ball goes in a direction other than intended or such that it looks like the agent is dribbling before it passes. These "errors" are important for the coach; it allows the coach observe other possible actions and results of the original advice.

We ran the MDP learning on the 90 games of data. Figure 4.8 shows the abstract state space we used, for both these experiments and the full game experiments discussed in Section 4.4.2. We used the Ball Arc player occupancy regions for this experiment. Here, since there were no opponents on the field, the total possible size of the abstract state space was 5882 states. The effective state space size (states actually observed) was 346 states. Our algorithms produced a Markov Chain and a Markov Decision Process from this data.

We experimented with adding different reward functions. For each reward function, reward was associated with *all* states in which the ball was in a particular grid cell (see Figure 4.7), namely:

Cell 13 The cell where the upper left player usually stands.

[1]Several small modification of the standard soccer rules were needed, namely stamina and offsides were turned off and dead balls were put back into play after 1 second.

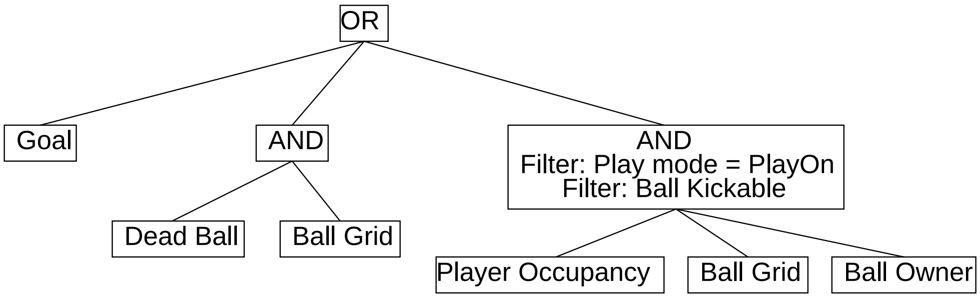


Figure 4.8: The abstract state space used in the simulated soccer environment environment. We used two different player occupancy factors for soccer. For the circle passing and some of the full game experiments, we used the Ball Arc regions, and for other full game experiments we used the Ball Path regions. Details of these regions are given in Section 4.3.2.

Cell 3 The cell immediately above the previous one. No agent would normally be here, but some passes and miskicks will result in agents being in this square.

Cell 34 Near the middle of the field. Agents tend to move towards the ball (especially to try and receive a pass) so agents often end up in this cell during training.

Cell 14 To the right of the upper left player. Since this cell is in the normal path between two players passing, the ball will frequently pass through this square. On a miskick, either one of the two closest agents could end up in that square with the ball.

One important note for all the reward states is that since the Ball Kickable filter is used, abstract states are only observed when an agent can kick the ball. This means that the ball merely passing through a region is not enough; one of the agents must touch the ball while it is in the region. Further, although the training data had the agents passing the ball in a circle, solving the MDP will lead the agents to get to the reward states as fast as possible. Therefore, the execution with advice will not replicate the circle passing, but achieve the specified reward quickly.

There are other reasonable reward functions. We chose these rewards to vary the degree to which the reward states were observed during training. Similar states around any one of the players would likely give similar results.

We ran with the CM3 agents receiving advice in each of these scenarios. Actions were sent if they were within 99.9% of optimal. We randomly chose 100 spots in the area around the players and for each of the eight cases (4 different rewards and training/with

MDP advice) put the ball in each of those spots. The agents then ran for 200 cycles (20 seconds). A trial was considered a success if the agents got to any reward state in that time. The time bound is somewhat arbitrary, but varying the time bound somewhat does not significantly affect the relative results. Further, the completion results at a particular time are easier to present and discuss than the full series of rewards received. Table 4.3 shows the results for the agents executing the learned advice for the four different reward functions. The "Training Success" line shows the percent of trials which completed with the specified reward during the initial UTA training games as a basis for comparison.

Reward Grid Cell	13	3	34	14
# times reward states observed	5095	211	1912	2078
Training Success %	53%	4%	40%	44%
Advice Success %	77%	21%	88%	69%

Table 4.3: Performance for circle passing. The reward state counts are the numbers of times reward was given during training.

In all scenarios, the success percentage is higher with the MDP based advice. The success percentage observed correlates well with the number of times the reward was observed in training for two reasons. First, the more paths to the reward state that are seen in training, the more likely that the coach has useful advice for any state that actually arises. Second, the more often the reward state was seen, the easier the reward was for the agents to achieve even without advice.

Table 4.3 ignored how long it took to achieve the reward. Figure 4.9 shows the percentage of trials which achieved reward for a variety of times. The graphs do not reveal any particular time bound which makes a large difference, justifying our use of percent completion by a given time bound as a measure of the advice's effectiveness.

In all cases we see that agent execution is not perfect. This occurs for several reasons. Noise in the perception and execution can cause actions to fail and have undesired effects. Execution of some abstract actions (such as passing) require more to be true about the underlying states than what is expressed by the abstraction. In passing, another agent needs to be in or near the target location in order to receive the pass. Therefore, it can happen that the advised abstract action can not be done at this time.

The reward cell 3 scenario was the most difficult for the agents to achieve. It was also the set of reward states which was least often seen in training. More examples of transitions to reward states should allow a better model to be created.

These experiments demonstrate that the coach can learn a model from observation

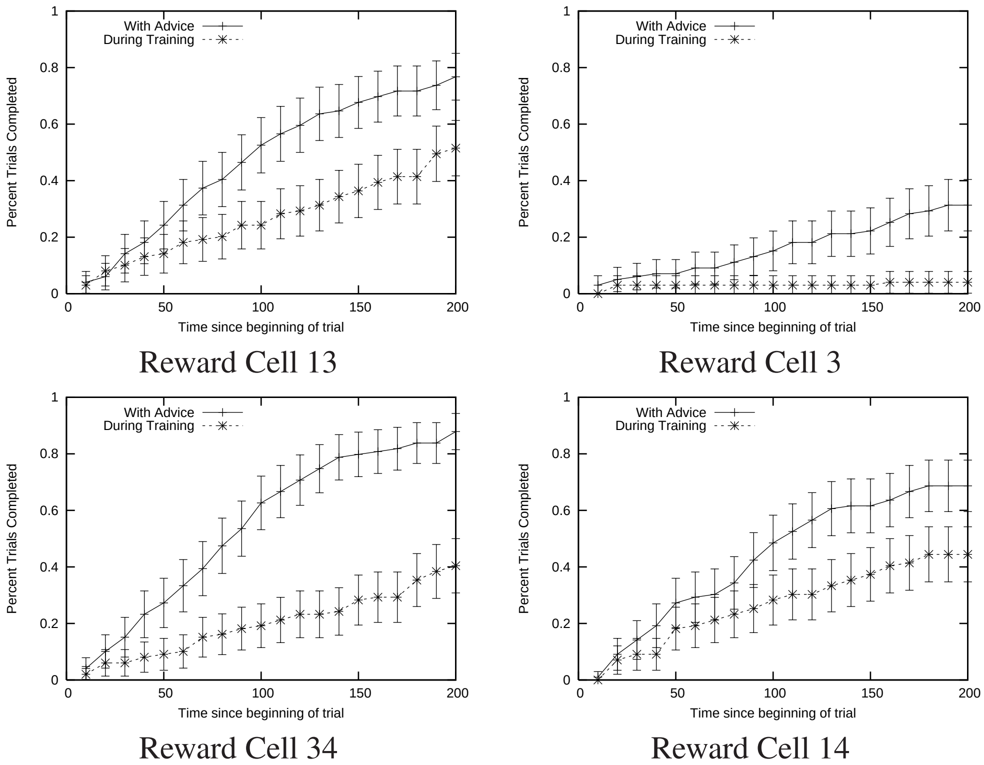


Figure 4.9: Time taken for circle pass trials in testing and training. The x-axis is the time since the beginning of a trial and the y-axis is the percentage of the trials which received reward by that time. The error bars are 95% confidence intervals.

which can then be used to improve agent performance. Changing the reward function can allow the same transition model to be used to generate different advice. However, the advice produced may not be that effective if not enough instances (relative to the noise levels in the world) were observed in the training data. Also, the training date came from the execution of one set of agents (UTA) and was successfully used to provide advice to a different set of agents (CM3). This fact demonstrates that the abstract action templates can be useful across agents.

4.4.2 Soccer: Full Game

We now move on to MDP learning and advice in the full soccer environment. It should be emphasized at this point how challenging the soccer environment is. The performance of a team is the product of many tightly integrated and interacting factors, such as:

- Teams must perform correct time management with the server. Noise inevitably arises here because the agents are subject to the whims of the scheduling algorithms of the operating system. However, systematic changes can drastically affect a team. For example, at RoboCup 2000, after setting up our team, I discovered that the agents' performance was much lower than expected (based on test games against known opponents). After much further testing, it was discovered that agents were (on average) sending actions to the server 40ms after the beginning of the simulation cycle rather than 60ms (which was done in previous testing). This small change greatly affected the effectiveness of the agents.
- While essentially all teams implement the same basic set of skills like passing, dribble, and shooting, the performance characteristics can vary greatly. Some teams dribble faster and some safer. Some teams shoot quicker and with less power and some do the opposite. In addition, decisions about where to look in the world greatly affect the responsiveness of the agents. None of these properties of the performance of the skills can be directly affected by the coach.
- Abilities and configurations of the above skills affect greatly the effectiveness of various strategies. For example, a team that dribbles well may not need to position players forward of the ball to receive forward passes. Teams that intercept the ball quickly may not need to stay as close to opponent players to defend against them.
- Optimal policies are completely unknown for the environment. While much research has gone into creating effective action policies, no one can provide a good estimate for how close to optimal any policy or team is.

- The results of a game have a large amount of noise. In many cases we will be looking at the mean score difference in sets of games. Across all of our experiments the median of the mean score difference in a set of games is 6 goals. In other words, the average experiment run of 30 games has a mean goal difference of 6. However, the median of the *range* of score differences in a set of games is 9 goals. In other words, in the average experimental run of 30 games, the observed score difference will vary by 9 goals, which is larger than the observed mean score difference. The distribution of scores is very wide compared to the magnitude of the score difference statistic.[2] In addition to making experimentation more difficult, this randomness complicates the learning process of the coach since full game effects must be observed quite often in order to be reliable.

Factors such as these make any improvements through advice in overall performance (as measured by score) a noteworthy achievement.

Opponent with Flaw

One ability that we would like a coach to have is to exploit the flaws in an opponent team. If the training data is from that flawed team playing, then our MDP learning system should be able to model and then exploit some flaws of an opponent team. In this section we discuss experiments playing against an opponent with a particular flaw we designed.

We took the CM4 agents and modified them such that they will avoid going into a corridor down the middle of the field. We will call this team FLAW. The corridor is the rectangle shown in the pictures in Figure 4.10.

The training data for the MDP learning was 100 games of 3 different teams:

- 30 games of SM vs. FLAW
- 30 games of EKA vs. FLAW
- 40 games of CM4 vs. FLAW

An MDP was learned using the state space from Figure 4.8 with the Ball Path player occupancy regions (see Section 4.3.2).

[2]An alternative way to think about this is that the median standard deviation is 2.3, so 95% of the score differences of the games should lie in ± 4.6 goals of the mean score difference.

Table 4.4 shows some game statistics for the various teams in training and CM4 when being coached with the learned MDP. Note that the CM4 team has a lower overall performance (on all statistics) compared to both SM and EKA, suggesting that CM4 is a weaker team overall. Since many factors that can not be changed by the coach's advice go into making a strong team, we can not expect CM4 to reach the same performance as SM and EKA. On all statistics, CM4 is helped significantly by the advice from the learned MDP.[3] The performance of SM and EKA are provided as a point of comparison.

Teams	Score Difference	Mean Ball X	% Time Attacking
SM vs. FLAW	12.2 [11.3,13.2]	19.0 [18.93,19.11]	43%
EKA vs. FLAW	7.3 [6.5,8.1]	14.6 [14.47,14.65]	35%
CM4 (training) vs. FLAW	0.7 [0.4,1.0]	1.1 [1.04,1.16]	24%
CM4 (w/ MDP) vs FLAW	3.1 [2.5,3.7]	9.5 [9.46,9.64]	35%

Table 4.4: Statistics for training and testing against the flawed team. All intervals are 95% confidence intervals (the confidence intervals on the percent time attacking are all less than 1%). For score difference, positive is winning. For mean ball X, larger X indicates that the ball is closer to the opponent goal. Time attacking is the percentage of time that the team controlled the ball on the opponents' half of the field.

In addition to seeing a strong positive effect in the final score, we are interested in whether the learned MDP is producing advice which matches our intuition that the team should dribble and pass in that corridor more. Qualitatively, we can see this is true. Figure 4.10 shows four example games from testing and training. While there are instances in the training games of dribbling and passing through the forbidden corridor, it is much more common in the training games.

More quantitatively, Figure 4.11 shows a histogram of the differences of the ball X location (in the corridor) during training and testing. Each bar represents the number of cycles the ball was in the corridor for that X bin in testing minus the number of cycles during training. Therefore, a negative bar represents that the ball was in the X bin in the corridor less in testing and a positive bar represents that the ball was in that X bin in the corridor more in testing. The group of positive bars on the right hand side show that the coached agents are moving the ball to the right hand side of the corridor (towards the opponent goal) and consequently, spending less time on the left side of the corridor. This effect is what one would expect if the coach's advice directs the agent to move to the right hand side of the corridor when possible.

[3]Neither SM or EKA understand CLang, so we can not coach these teams.

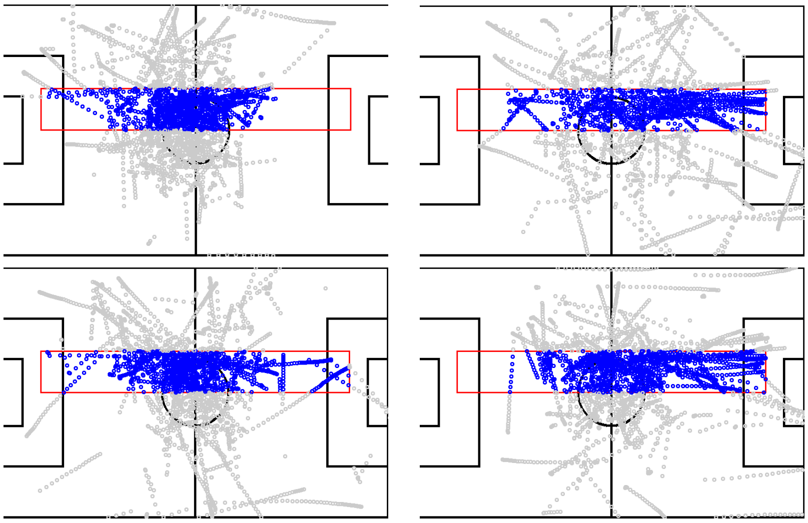


Figure 4.10: Locations of the ball throughout 4 sample games, two from training (CM4 vs. FLAW) and two from testing (CM4 coached with the learned MDP vs. FLAW). The rectangle is the area on the field that the team on the right (FLAW) avoids going into.

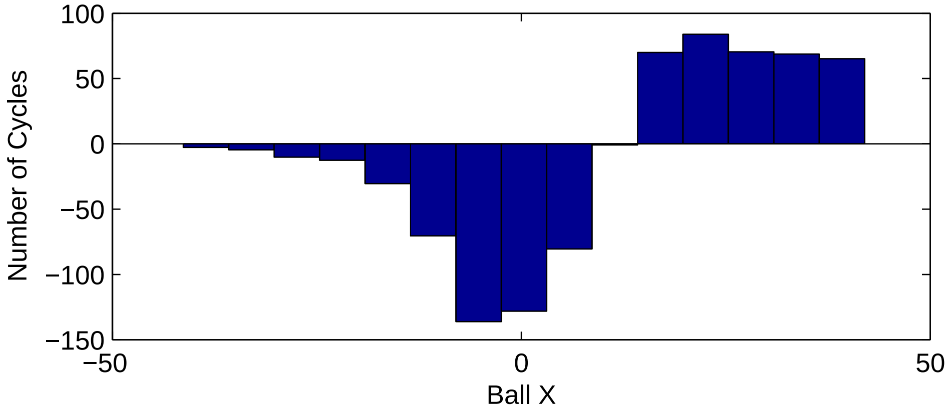


Figure 4.11: Histogram of the ball's X location when in the corridor. The right hand side is closer to the opponent's goal. The y-axis is the number of cycles per game (on average) that the ball was in the given X bin in testing minus the number of cycles per game (on average) in training.

Overall, this experiment demonstrates that the model learning is able to successfully identify and exploit a pattern of behavior in opponents. Since we do not have an *a priori* model of the soccer game, testing against a team with a known flaw allows use to verify that the model learning can find appropriate patterns in the execution data. We therefore conjecture that similar patterns can be extracted in other environments.

Various Opponents

Now that we have experimentally verified that the MDP learning can be used to create advice which exploits an opponent with a known flaw, we move on to experiments involving real teams in the full soccer game. We believe that the advice from the learned MDPs should have a positive effect on the play of the team. Additionally, we will compare MDPs learned from a variety of opponents to MDPs learned from playing against the actual opponent being tested against.

We experimented with a number of different learned MDPs. Table 4.5 provides a list of the MDPs with information about the parameters of the learning. One important point that has not been mentioned in that in looking at a logfile, the coach can analyze the game from both the left and right teams' perspectives (which is what was done for BallArc023 and BallPath5) or from just one team's perspective. The latter makes more sense for the MDPs which are learned from data against a particular opponent in order to learn only the transitions exhibited by that opponent.

Name	Player Occupancy	# games	Input
BallArc023	BallArc	2482	all past
BallPath5	BallPath	2482	all past
UVAvBH	BallArc	100	UVA playing BH, only using UVA perspective
UVAvEKA	BallArc	100	UVA playing EKA, only using UVA perspective
BHvEKA	BallArc	100	BH playing EKA, only using BH perspective

Table 4.5: Learned MDPs used in soccer game experiments. All MDPs use the abstract state space tree shown in Figure 4.8, but with different player occupancy regions. The "all past" inputs are all logfiles from RoboCup2001 and 2002, German Open 2002 and 2003, Japan Open 2002 and 2003, American Open 2003, and Australian Open 2003 (601 logfiles) and 1724 more logfiles from our previous experiments with a number of past teams (see Chapter 5).

We ran test games of the CM3 team running with various MDPs as shown in Table 4.6.

The baseline was the team running with a formation (as described in Section 5.2.1) and setplays (as described in Chapter 6).

Description	Baseline	BallArc023	UVAvOpp	BHvOpp
vs. BH	-7.5	-7.0	**-5.8**	
	[-8.30,-6.64]	[-7.94,-5.99]	**[-6.55,-5.12]**	
vs. EKA	5.1	**8.2**	**6.2**	**6.1**
	[4.32,5.93]	**[6.93,9.41]**	**[5.46,6.99]**	**[5.37,6.83]**

Table 4.6: Soccer game results of CM3 against several opponents with a variety of learned MDPs. Each entry is the score difference, where positive is the CM3 team winning the game. The intervals shown are 95% confidence intervals. The entries in bold are statistically significant differences compared to the baseline at the 5% level with a one-tailed t-test. The vOpp MDPs refer to the vBH or vEKA MDP as appropriate for the opponent.

The most important note about the results is that in spite of the great difficulty of the task, the learned MDPs are able to have positive effects on the overall score of the games. While not all MDPs result in significant positive effects against all teams (though note that no MDP results in a statistically significant negative effect), learning an MDP can help the team perform the task better in this relatively uncontrolled environment.

As shown, we experimented with learning MDPs specific to a particular opponent. The MDPs learned from data of the opponent playing (the UVAvOpp and BHvOpp columns in Table 4.6) perform significantly better than the baseline, but the results are mixed comparing to the non-opponent specific data. For the RCSSMaze environment, Section 4.4.3 further explores the dependence of the learning on the input data.

Coach Only Team

This section continues experiments dealing with the entire soccer game. Throughout, we will be using the BallPath5 MDP as described in Table 4.5. The team which will be coached with the MDP has the restriction that the team will only perform passes and dribbles that the coach recommends. This team is a modification of CM4 (see Table 4.2), and we will call it COCM4 (for Coach-Only CM4). This restriction is to both emphasize the coach's advice and to model a team which has very little knowledge of what to do. All experiments will be with the COCM4 team playing the CM4 team but using a variety of formations for the two teams.

Restricting the team in this way allows us to focus our evaluation on the value of the policy of the learned MDP. The previous experiments could only evaluate the policy in relation to the default actions/policy of the coached team. By removing some of the coached teams policy, the effect of the MDP based policy can be more clearly observed.

We use three different formations for both the coached and opponent teams, and play all possible combinations, with and without the learned MDP advice. Table 4.7 shows the results. In every case, the team using the learned MDP advice outperforms the team without such advice and goes from losing on average to winning or tied.

Coached COCM4	CM4 Team Formation		
team formation	Defensive(532)	Normal(334)	Offensive(3232)
Defensive(532)	-.27[-0.617,0.0841]	-0.33[-0.57,-0.097]	-0.47[-0.71,-0.22]
	.27[0.0192,0.514]	0.2[-0.018,0.42]	0[-0.24,0.24]
Normal(334)	-0.23[-0.57,0.10]	-0.1[-0.32,0.12]	-0.8[-1.14,-0.46]
	0.1[-0.096,0.30]	0.2[-0.056,0.46]	0.26[0.0020,0.53]
Offensive(3232)	-0.33[-0.59,-0.070]	-0.3[-0.57,-0.032]	-0.63[-0.95,-0.31]
	0.37[0.13,0.61]	0.47[0.24,0.69]	0.1[-0.12,0.32]

Table 4.7: Soccer game results for a number of variations of COCM4 playing against CM4. The rows represent different formations for the coached COCM4 team, and the columns represent different formations for the opponent CM4 team. The numbers represent standard soccer names for the formations and can be ignored for the non-soccer inclined. The top line in each cell is the baseline (no MDP based advice) and the bottom line is with the MDP based advice. The MDP used in every case was the BallPath MDP. All differences between the two entries in table cell are significant at the 5% level according to a one-tailed t-test. In all cases, the coached COCM4 team was constrained to only perform passes and dribbles recommended by the coach.

These experiments demonstrate again that the learned MDP is useful for providing advice to a team of agents in the soccer environment. The advice allows the team to win matches that they were previously losing. Further, these experiments provide evidence that the value of the advice comes at least in part from a good policy being extracted from the model and not just good selection/interpretation of advice by the agents.

4.4.3 RCSSMaze

We now move on from the soccer environment to the RCSSMaze environment (described in Section 2.3). Here, we can get cleaner experimental results, vary the environment and rewards more easily, and experiment further with how the source of input data affects the usefulness of the MDP.

All of these experiments will be verifying that the model learning algorithms are sufficient to learn models which are adequate representations of the environment. The models are shown to be adequate because an agent agent receiving advice from the model is able to greatly improve its performance.

All of the experiments in this environment are run in a similar way. The training data consists of 40 ten minute executions in the environment (what the agent is doing during the training will vary). The coach analyzes the logfiles and learns an MDP with the abstract state space shown in Figure 4.12. The state space is set up so that state 0 is the normal goal state (the leftmost branch of the tree) and the last state is the initial state (the rightmost branch of the tree). A reward of 100 is assigned to some (set of) states and a reward of -1 assigned to the initial state.

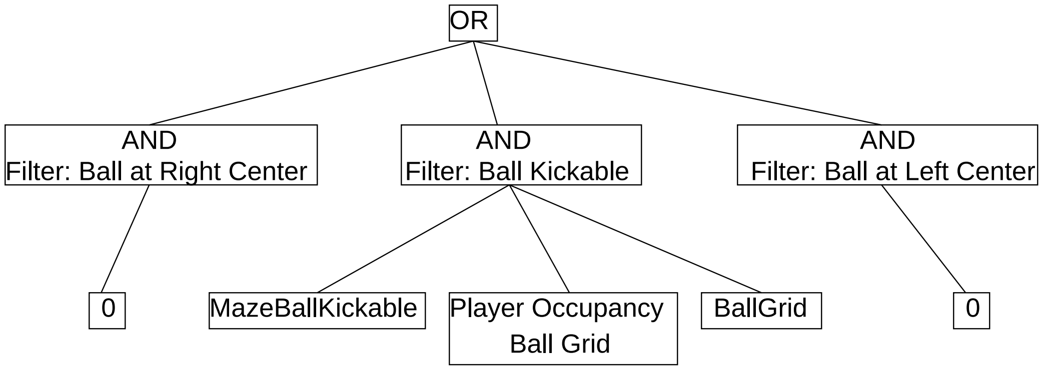


Figure 4.12: The abstract state space used in the RCSSMaze environment. The "0" boxes represent constant factors which are always 0.

In testing, in order to evaluate how well the agent is doing, we count the percentage of trials which are successful. A trial begins when the agent transitions from the start state and ends when either the agent achieves the positive reward or is reset to the start state (which usually means that the agent ran into a wall).

First, we consider maze 0. An important question is what the training data for the MDP should be. In order for the coach's learned MDP to be useful, a transition to a goal state must be observed at some point. Otherwise the goal state will be unconnected to the rest of the MDP and the value function will be negative everywhere. We initially tried to have the agent pick a spot 10m away and move until it got there (or until it got reset to the start state), but this did not result in the agent ever achieving the reward. We therefore chose seven special spots on the field, as shown in Figure 4.13. 95% of the time the agent needed to chose a new target to move to, it uniformly randomly choose one of those targets. The other 5% of the time, it chose a random point 10m away.

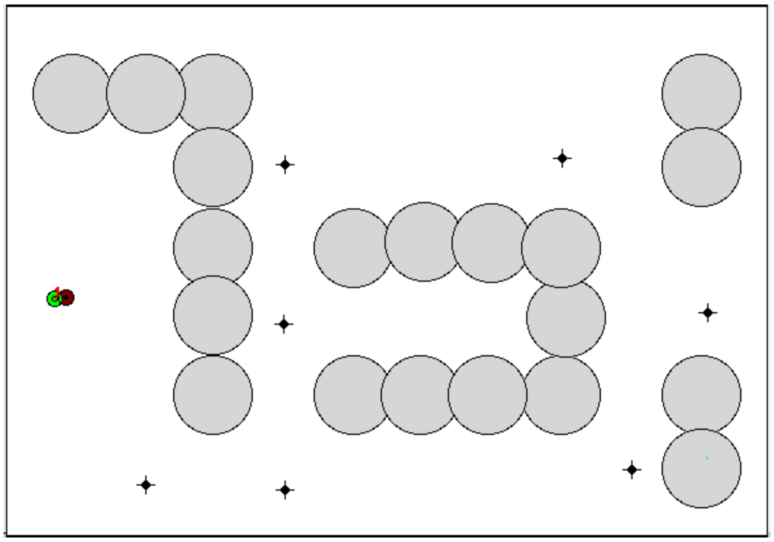


Figure 4.13: The agent targets for RCSSMaze 0 training. The 7 targets are marked by plus symbols. The agent is shown on the left middle.

Forty ten minute executions were run in this scenario. For the MDP learning, we consider three different (sets of) reward states:

Reward 0 Reward of 100 for getting to the far right center

Reward 1 Reward of 100 for running into a wall in the upper space

Reward 2 Reward of 100 for running into a wall in the lower space

In all cases we set a reward of -1 for being in the far left center, which is the start state that the agents gets reset to when it runs into a wall.

MDPs were learned for each of these rewards and then forty ten minute executions were run with the MDP advice. Table 4.8 shows the results. As expected, some reward scenarios are more difficult to achieve; the longer the path to the goal (as reflected in a lower number of overall trials), the more possibility for a failure, leading to a lower

	Training		Testing	
Reward	Total Trials	% Successful	Total Trials	% Successful
0	1620	1%	1060	64%
1	1620	1%	879	60%
2	1623	7%	1137	93%

Table 4.8: For the RCSSMaze 0, in the three different reward scenarios, trial success percentages for training and testing when running with an MDP learned for that reward.

success percentage. However, the models learned allow the coach to provide advice to the agent to greatly improve its reward.

Next, we consider maze 1. This maze is much more challenging for the agent. The correct action depends more strongly on the changing world state. For the upper route, the agent has to notice and wait for various agents to move. For the middle route, the agent has to do the same, but has the additional benefit of some wall moving as long as the agent stays in the same place.

We once again do training with fixed target move points, as shown in Figure 4.14. In addition, to capture the necessity of the agent waiting in place for the world to change, the agent simply holds in place for 40 cycles out of every 200 cycles.

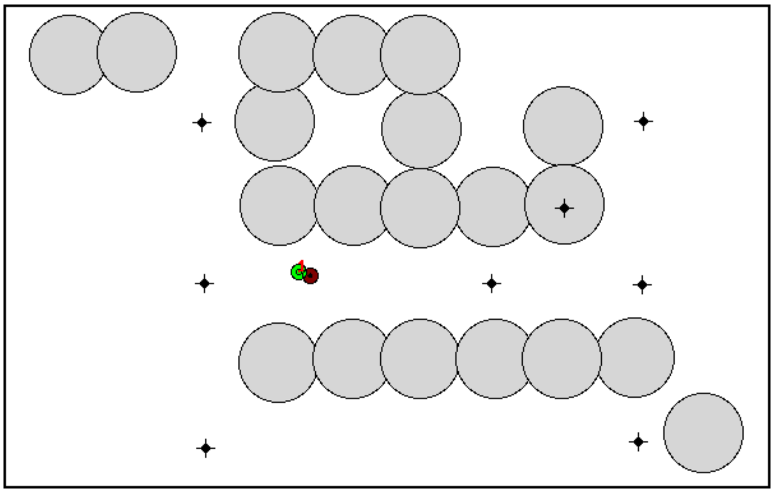


Figure 4.14: The agent targets for RCSSMaze 1 training. There are 8 targets marked by plus symbols. The agent is shown in the middle of the figure and the larger circles are walls.

After running the forty ten minute executions for training, we assign a reward of 100 for getting the ball to the far right center and a reward of -1 for the initial state at the far left center. Note that there are two possible ways for the agent to get to the positive reward. The upper route requires the agent to possibly wait for three different agents to move while the middle one requires waiting only once, but probably for a longer time.

Using the MDP, the agent successfully completes 69% of the 945 trials. There are two interesting points about the executions. First, the advice sends the agent through the middle channel. While frequency in training does not necessarily reflect what will be optimal in the MDP, it should be noted that 39 of the 44 times that reward was received in training were through this middle channel. Second, if the agent goes down the middle channel and there is a wall blocking its way, the moves to the left and then to the right again. This behavior deserves some further explanation.

With our global knowledge, what do we believe is the best behavior for the agent when it proceeds down the middle corridor and discovers that its way is blocked? This situation is depicted in Figure 4.15. It would seem that the agent should just wait until the wall moves out of the way. Instead, the MDP advises the agent to back up to the grid cell one to the left, then move forward again. While this is a little surprising, it turns out to work well. This behavior reveals clearly two of the assumptions in the learning:

- The world is assumed to be Markov *from the perspective of the abstract state space*. If the Markov assumption is broken, then the model can produce unusual or incorrect advice.

  In the maze environment, when the agent is in the left grid cell of Figure 4.15, the abstract state space does not include the information about whether the way ahead is blocked. When the agent moves into grid cell on the right, the way will be either blocked or not. From the MDP perspective, this will occur randomly. Since the wall spends equal time blocking and not blocking the corridor, the probabilities will be approximately equal. At the MDP level, if the "blocked" transition occurs, it makes sense to just back up and try the "move to the right grid cell" action again. Because of the assumed Markov property, whether the wall is blocked the next time is assumed to be unrelated to whether it was blocked this time.

- The MDP is not modeling the time taken by actions. If all actions take approximately the same amount of time, this may not matter much. However, if some actions take significantly longer than others, the learned model can produce unusual or incorrect advice.

  In the maze environment, staying in place leads to abstract states being observed every cycle while moving leads to less frequent abstract states. Therefore, from the

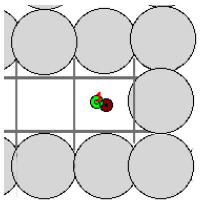


Figure 4.15: Detail of the decision faced by the agent when blocked by a wall in the middle corridor of maze 1. The grid lines correspond to the coach's grid as shown in Figure 4.3. The agent is depicted in the middle of the figure and the walls are the large circles.

> MDP perspective, the holding action result in more observed state transitions while in actuality taking the same amount of time. Other abstract action formalisms, such as Options [Sutton et al., 1999], provide time estimation and combining the learning algorithms here with these kinds of formalisms is an interesting direction for future work.

In summary, despite the minor violations of assumptions, the MDP learning is able to find an effective policy for this more challenging maze environment as well.

The last problem considered experimentally in this environment is using the execution of one MDP as training data for another. One of the benefits of the model learning algorithms is that various rewards can be applied to the same model. However, what effect does changing the training data have on the ability to learn a useful model?

We return to considering maze 0. Originally, we used training data based on the agent randomly going to fixed points as shown in Figure 4.13. Instead of using this training data for an MDP, we now consider executions of advice from a previously learned MDP as training data for a new MDP. We use exactly the same process as before, just with different input observations.

We have three additional sets of training data, one from each of the executions of the MDP learned for rewards 0, 1, and 2. From each of these, we can theoretically learn an MDP for rewards 0, 1, and 2.

Results can be seen in Table 4.9. First, note that several entries in the table are marked "n/a." A useful MDP can not be learned from training data if a reward state is never encountered. In only one case (reward set 0 from reward set 1) was a reward from a different reward set ever observed during execution using the previous MDP. In that one case, a reward set 0 state was observed exactly once, and as can be seen, the resulting

MDP is not useful for getting the agents to achieve that reward again.

		# Rew. Seen in Training			% Success in Testing		
Training Data	# States	Rew. 0	Rew. 1	Rew 2	Rew. 0	Rew. 1	Rew 2
Original	371	11	115	1055	64%	60%	93%
From Reward 0	257	676	0	0	82%	n/a	n/a
From Reward 1	243	1	2909	0	0%	67%	n/a
From Reward 2	147	0	0	9088	n/a	n/a	78%

Table 4.9: Results for learning a new MDP from a previous MDP execution in RCSS-Maze 0. All differences in the percent success columns between entries in the same column are statistically significant at the 5% level.

Next, for the reward set 0 and reward set 1 cases, using the training data from a previous MDP testing improves the effectiveness of the MDP. The agents have successful trials 18% and 7% more often using the new MDP.

The reward set 2 case is quite different. The MDP learned from the original data performs better than the MDP learned from the execution traces of the first MDP. By comparing the MDPs, we find that the second MDP is a victim of the success of the first. Of the 308 unsuccessful trials, 245 of them go through two particular states. Both of these states are immediately to the right of the starting state (grid cell 21) with different wall occupancies. In contrast, only 27 of the 1102 successful trials go through these states. In the original MDP, the recommended action was to move down (to grid cell 31). In contrast, in the second MDP, the null action has the highest value (with the move action second). In this second MDP, the null action transitions to the start state with probability 1. The start state has a high value despite its negative reward of -1 because, in the training data, the many examples of successful trajectories to achieve the +100 reward make the probability of those transitions very high. In other words, the distribution of actions that produced the transitions was very skewed and therefore the probabilities estimated by the MDP do not match what is actually observed. This skew of the distribution of observed actions is occurring in the other recursively learned MDPs, but does not have the negative effect observed in reward set 2.

These experiments suggest that the source of the observations used for learning can significantly affect the efficacy of the MDP. In particular, if states leading to the reward states are not observed often enough (or at all), the model will not have sufficient fidelity to produce a useful policy/advice. One of the many interesting avenues opened by this thesis is a further exploration/formalization of this dependence. Further, the possible increase or decrease in performance as a result of tailoring the input data specifically to the task mirrors the effect seen in the soccer environment in Section 4.4.2.

4.5 Summary

This chapter has presented a set of algorithms for learning an environment model (a Markov Decision Process) from a set of observed agent executions in the environment. The observations notably do not include information about what actions the agents were executing. Rather, action information is inferred from supplied templates and observed frequencies. Use of learned MDPs to provide effective advice to agents is demonstrated in several variations of the simulated soccer environment and the RCSSMaze environment. The experiments also reveal that if the Markov assumption on the abstract state space is violated, unusual behavior can result. Additionally, learning a new MDP from execution of a previous MDP can produce advice which is either more or less effective than the previous MDP.

Chapter 5

Coaching in the Presence of Adversaries

Chapter 4 presented algorithms for a coach to learn about an environment in order to provide advice. In environments with adversarial agents, learning about and adapting to adversary behavior has great potential for helping a team. Learning about adversary agents is an alternative source of advice compared to learning about the environment, though the approaches can be complementary. Together, these approaches provide the knowledge sources for advice in the breakdown of the coaching problem discussed in Chapter 1.

This chapter discusses several opponent model representations and learning algorithms for a coach agent to learn about an adversary and use this knowledge to give advice to agents. Empirical validation is performed in the simulated robot soccer domain.

5.1 Probabilistic Movement Models

The team planning for distributed execution (Chapter 6) requires that the coach be able to predict the movements of the opponent players in order to evaluate the safety of a plan. This section describes the representation, use, and selection of opponent models for this planning.

The opponent models we present will specifically model opponent movement and position in a two dimensional plane, though the representation and algorithms should easily extend to higher-dimensional metric spaces. We use the models to evaluate the quality of various possible planned actions, but the same models would be useful for any application where prediction of opponent movement would be useful.

Our work on opponent modeling is driven by the goal of improving the performance of teams of agents through their adaptation and effective response to different adversarial teams. In the past, work has been done on adapting the teammates' behavior to their opponents' behavior, mainly at the individual low level of positioning and interactions between a small number of agents [e.g., Veloso et al., 1999, Stone et al., 2000]. While such schemes do behave differently when paired with different adversaries, there is no long term adaptation. Every time the adversary takes the same series of actions, the team responds in the same way.

We specifically focus on situations where the game is stopped. At these times, a team can coordinate and come up with a plan of action to execute once the ball is back in play. These plans are known as setplays. In the standard rules for the simulated robot soccer environment, this is also one of the times that the coach is permitted to talk (see Section 2.2.1). Therefore the coach can make a plan which is then executed by the distributed agents. Several preset setplay plans have been introduced that provide opportunities to position the teammates strategically and have been shown to contribute to the performance of a team [Veloso et al., 1998, Stone et al., 1999, 2000]. Here, we contribute *adaptive* setplays that change and improve throughout a game in response to the opponent team's behavior. We address the challenging problem of capturing some essential information about the opponent team's behavior to provide this adaptation.

Throughout, we will not be reasoning about any modeling that the opponents do of our team. This choice is primarily for computational tractability and simplicity; handling recursive agent modeling can be quite challenging [Gmytrasiewicz and Durfee, 1995]. Also, the amount of data we have to work with about a particular opponent is quite small. If we considered richer opponent models involving recursive modeling, we would need more data to correctly recognize the models.

Because of the short time span in a simulated soccer game, we decided to begin a game with a fixed set of models and choose between them during the game. Selecting a model should require fewer observations than trying to create a model from scratch.

These models are not intended to capture the full strategy or movements of the opponents. Rather, the models only need to capture the way the opponents move in the two to twenty seconds after the game has stopped, while the setplay is actually going on.

Two assumptions related to the opponents should be noted. First, for best effectiveness, these models should have good predictive power with respect to the set of opponents expected. We assume that the variation in opponents can be approximately expressed in a reasonably sized set of models from which to choose.

Second, the output of an opponent model does not *explicitly* depend upon the positions of our players. However, the output does depend on the anticipated path of the ball, which the position of our teammates also depends on. The choice is made for computational simplicity, especially in the context of the plan generation discussed in Section 6.3. No fundamental reason would prevent a model from taking our teammates' positions into account and this is a direction for future work.

We will present the model representation and selection algorithm in general form in Sections 5.1.1 and 5.1.2 and then discuss their use in robot soccer in Section 5.1.3. Section 5.1.4 then presents an experimental validation.

5.1.1 Model Representation

Opponent models often have the general form of a function from state to actions (perhaps a probability distribution over actions), which are intended to predict the actions the opponents will take [e.g., Carmel and Markovitch, 1998]. We will follow this same basic strategy, but predict the resulting state of the opponents rather than the actions taken to get there. We also consider predicting the behavior of a team of agents, rather than a single agent.

Let p be the number of players on a team. For notational simplicity, we will assume our team and the opponent team have the same number of players, but this is not essential to our formulation. We assume that the state of the world can be decomposed into three components:

$\mathcal{S}_T^p$ This sequence of p elements represents the state of each of our team members. In other words, each agent's state can be represented as an element of $\mathcal{S}_T$.

$\mathcal{S}_O^p$ The states of the p members of the opponent team.

$\mathcal{S}_W$ The state of the world not represented by $\mathcal{S}_T^p$ or $\mathcal{S}_O^p$.

This state decomposition into agents' states is similar to that used by Xuan et al. [2001] to model communication between agents.

Let $\mathcal{R}_O$ represent the set of probability distributions over $\mathcal{S}_O$ and let $\mathcal{A}$ represents the set of sequences of possible actions (including durations, if applicable) our team can take. Conditional plans, where the next action depends on the observed state, are not considered here.

An opponent model is then a function which probabilistically predicts the opponents' future states based on the world state, opponent states, and planned actions of our team.

In other words, we define a model M as a function:

$$M\colon\ \mathcal{S}_W \times \mathcal{S}_O^p \times \mathcal{A} \to \mathcal{R}_O^p \tag{5.1}$$

Note that our team's state $\mathcal{S}_T^p$ is not part of the inputs to the model. However, the actions of our team $\mathcal{A}$ are constrained by the team's states so some such information is available to the model. The choice to remove the explicit dependence on $\mathcal{S}_T^p$ is made for computational simplicity. On the other hand, each opponent player's final state distribution may depend on the starting states of *all* the opponent players.

An opponent model of this form can be used to calculate the probability of an opponent ending in a particular state. In particular, given a world state $w \in \mathcal{S}_W$, opponent states $s_i \in \mathcal{S}_O$ $(\forall i \in [1,p])$, and a planned team action $a \in \mathcal{A}$, an opponent model M says that the probability for player j being in ending state $e_j \in \mathcal{S}_O$ is

$$P_j[e_j|w, s_1, \ldots, s_p, a, M] := M(w, s_1, \ldots, s_p, a)[j](e_j) \tag{5.2}$$

We use P_j to represent the probability over ending opponent states for opponent j. Notationally, we also consider probability distributions to be functions from the input set to the real numbers.

In contrast to opponent models for game tree search [e.g., Carmel and Markovitch, 1996], our opponent models are predicting not just one action response of an opponent, but the result of a series of interleaved team and opponent actions. Our model is explicitly operating on abstract temporal and action levels, making it more applicable to environments with continuous or many discrete action opportunities.

5.1.2 Model Selection

Given the description of the opponent models, we can now describe the algorithm for selecting the best matching model. Given the assumption that the opponent has chosen one of our models at the beginning of the game and is then independently generating observations from that model, we can use a naive Bayes classifier.

We maintain a probability distribution over the models. The original distribution (the prior) is set by hand. Then, whenever a planning stage is entered, the model with the highest probability is used. Upon observing a plan execution, we use observations of that execution to update our probability distribution over the models.

We start with a probability distribution over the set of models $\{M_1, \ldots, M_m\}$ and then observe. An observation is a tuple of starting world state $w \in \mathcal{S}_W$, starting states for all the opponent players $s \in \mathcal{S}_O^p$, a planned team action $a \in \mathcal{A}$, and ending states for all opponent players $e := \langle e_1, \ldots, e_p \rangle \in \mathcal{S}_O^p$. We want to use that observation to calculate a new probability distribution, the *posterior*. That distribution then becomes the prior for the next observation update.

Consider one updating cycle with an observation $o = \langle w, s, a, e \rangle$. We want $P[M_i|o]$ for each model M_i. Using Bayes' rule we get

$$P[M_i|o] = \frac{P[o|M_i]P[M_i]}{P[o]} \tag{5.3}$$

We make the following assumptions in order to simplify equation (5.3).

1. The players movements are independent. That is, the model may generate a probability distribution for player x's ending state based on everyone's starting states. However, what the actual observation is for player x (assumed to be sampled from this probability distribution) is independent from the actual observations of the other players.

2. The probabilities of a particular set of starting states and planned action are independent of the opponent model. This assumption is questionable since the planned agent actions may depend on the opponent model determined to be the most likely. However, results in Section 5.1.4 demonstrate we still are able to recognize models correctly.

$$\begin{aligned}
P[M_i|o] &= \frac{P[w,s,a,e|M_i]}{P[o]} P[M_i] \quad \text{(from eq. (5.3))} \\
&= \frac{P[e|w,s,a,M_i]P[w,s,a|M_i]}{P[o]} P[M_i] \\
&= P[e|w,s,a,M_i] \frac{P[w,s,a]}{P[o]} P[M_i] \quad \text{(assump. 2)} \\
&= \underbrace{P[e_1|w,s,a,M_i]P[e_2|w,s,a,M_i] \ldots P[e_p|w,s,a,M_i]}_{\text{what opponent model calculates (eq. (5.2))}} \\
&\qquad \underbrace{\frac{P[w,s,a]}{P[o]}}_{\text{norm. constant}} \underbrace{P[M_i]}_{\text{prior}} \quad \text{(assump. 1)}
\end{aligned} \tag{5.4}$$

The term labeled "norm. constant" is a normalization constant. That is, it does not depend on which model is being updated, so we don't have to explicitly calculate those terms. We calculate the remaining terms and then normalize the result to a probability distribution.

The preceding computation began with the assumption that the opponent has chosen one of our models and was generating observations from it. Of course, this is only an approximation. However, if one model generally makes better predictions than the others, then that model will be the most likely. Assuming one of the given models is correct is the same type of assumption that is made in a number of statistical machine learning problems. The question we are trying to answer is "Which of the models from this set best explains opponent behavior?" This parallels the question in most machine learning tasks with generative models of "Which hypothesis from this set best predicts the data?"

In addition to the update above, we use weight sharing at the end of each update cycle. A small probability mass (0.1) is added to the probability value for every model and then the distribution is renormalized. This means that if there are m models, a probability p becomes:

$$\frac{p + 0.1}{1 + 0.1m} \tag{5.5}$$

Weight sharing prevents any model's probability from going arbitrarily close to 0, while not changing which model is most likely on any one update. Weight sharing allows the update process to more quickly capture changes in the opponents behavior (if their behavior switches from one model to another). For example, if the prior and an observation tell us that one model has probability of 1, we still put a probability mass of $\frac{1}{10+m}$ on every other model, meaning that we still believe there is a chance that the opponent team will, in the future, act as described by the model.

Weight sharing also means that more recent observations are weighted more heavily. Each step of weight sharing smoothes out the probability distribution. The perturbation caused by an observation (i.e. making one or more models more likely based on what was observed) is smoothed out by the weight sharing steps of other observation updates. More recent observations have gone through less smoothing operations and can therefore have an effect of larger magnitude.

If the opponent is changing or adapting, the coach may be able to track the changes, depending on the speed of adaptation. Of course, a team that knew exactly the algorithm we were using could still conceivably adapt just faster than the coach could keep up with. Applying regret-minimization techniques such as Auer et al. [2002] is an interesting future direction, but as far as the author knows, the complexity of the environment prevents the direct application of any known techniques.

5.1.3 Models in Robot Soccer

Conceptually, we want an opponent model to represent how an opponent plays defense during setplays. We conjecture that a wide range of decision making systems of the opponent can be roughly captured by a small set of models, though we have not empirically verified this conjecture.

Remember that p is the number of players on a team. Let $\mathcal{L}$ be the set of positions on the field, discretized to 1m. The player state sets $\mathcal{S}_T$ and $\mathcal{S}_O$ are both equal to $\mathcal{L}$, representing to location of a player. The world state $\mathcal{S}_W$ will also be equal $\mathcal{L}$, representing the location of the ball.

The planned ball movement will be the planned actions of our agents. We represent the ball movement as a sequence of locations on the field (an element of $\mathcal{L}^*$). The expected time for each ball movement, with bounds for normal execution, can be calculated based on the empirically determined environment and agent execution parameters, such as time to kick the ball and speed of the ball when passed.

An opponent model is then trying to predict where each opponent will be given the ball's current location ($w \in \mathcal{L}$), the opponents' initial positions $s \in \mathcal{L}^p$, and a future path of the ball $a \in \mathcal{L}^*$. In other words, the model answers a question like "Given the positions of the opponents and the ball, if the ball moves like this over the next 2 seconds, where will the opponents be at the end of those 2 seconds?" Formally, we have:

$$M : \underbrace{\mathcal{L}}_{\substack{\text{ball}\\ \text{position}}} \times \underbrace{\mathcal{L}_O^p}_{\substack{\text{opponent}\\ \text{starting}\\ \text{positions}}} \times \underbrace{\mathcal{L}^*}_{\substack{\text{planned}\\ \text{ball}\\ \text{movement}}} \rightarrow \underbrace{(\text{Probability Distributions over } \mathcal{L})^p}_{\substack{\text{predicted}\\ \text{opponent}\\ \text{positions}}} \tag{5.6}$$

This definition is an instantiation of Equation (5.1) for robot soccer. An example application of an opponent model is shown in Figure 5.1. Here, the model predicts that both opponents move towards the final location of the ball.

Thus far, we have described the format of the opponent model, i.e. *what* must be computed, but not *how* this computation is done. For the implemented system, all player distributions are represented as Gaussians. The models are simply functions which manipulate these distributions in the appropriate way. However, note that the selection algorithm described in Section 5.1.2 does not depend on this representation.

As an example, one of the models we use has all opponent players moving towards the ball. The function which represents the model adjusts the input distributions by moving

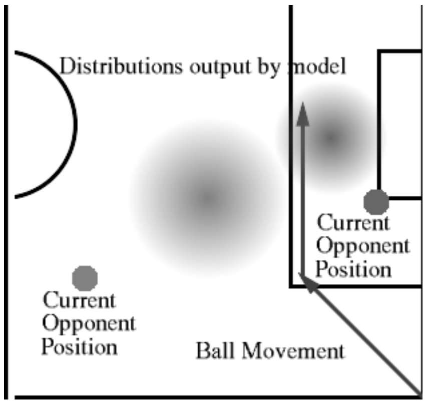


Figure 5.1: An example application of an opponent model. The fuzzy areas represent probability distributions for the two ending locations of the opponent players (shown as dark circles) as the ball moves along a path indicated by the arrows.

the means towards the ball and increasing all the variances. Section 5.1.4 discusses the set of models we use for the empirical validation in more detail.

In addition, a decision must be made about how to generate observations from the stream of data being received about the opponent positions. Clearly, if an observation tuple is generated every cycle we will be violating the independence assumption of the naive Bayes update, as well as giving the models little information (in terms of the ball movement) with which to work. On the other hand, the more observations the coach gets, the easier it is to correctly identify the correct model. To balance these competing factors, we decided to create an observation for processing every time the agent who is controlling the ball changes. An agent is considered to be controlling the ball if (i) the agent is the closest player to the ball and (ii) the agent can kick the ball. A given observation can cover anywhere from approximately 5 to 50 cycles (one half to five seconds) of movement.

5.1.4 Empirical Validation

The algorithms described above are fully implemented in the simulated robot soccer environment. This section describes experiments to validate their effectiveness.

A natural question to ask now is whether the opponent models described can be actually recognized by the algorithm we present. We use several assumptions during the probability updates, and if those assumptions are extensively violated, the recognition algorithm will fail to work as predicted.

In order to test whether these assumptions are violated, we first need to create a set of opponent models to use. In all of the models, the distribution of each player's final position is represented by a 2-dimensional Gaussian with equal variance in all directions. The standard deviation is an affine function of time (since the beginning of the setplay). The mean is computed as discussed below.

We created five models for the empirical evaluation. This set of models represent fairly distinct styles of movements by the opponents. The mean of each player's final distribution is computed relative to the initial position as follows:

No Movement At the initial position of the player

All to Ball Moved towards the ball at a constant speed

All Defensive Moved towards the defensive side of the field at a constant speed

All Offensive Moved towards the offensive end of the field at a constant speed

One to Ball This model is slightly different from the others. The ball's movement is broken down into cycles. At each cycle, whichever player is closest to the ball is moved 0.6m closer to the ball.[1] Note that since the ball can move faster than the players, which player is closest to the ball can change several times during the ball's movement. The final positions of the players are the means of the distributions.

It should be emphasized that these models are not for the opponent's behavior throughout an entire simulated soccer game. The models are only intended to capture the way the opponents move for the 5–20 seconds in which our team executes a set play from a dead ball situation. The models do not capture any actions that the opponents take with the ball, or how they play defense more generally.

The models are abstractions over player movements and we would like to verify that these models can be used to recognize the differences in teams. Further, we want to explore how long it takes for the naive Bayes based recognition algorithm to identify the correct model. The coach is the only agent doing the recognition since it has the global view of the field.

Before looking at how well each model can be recognized out of this set of models, we must first understand how well any recognition algorithm could expect to do. We call this the "separability" of a set of models. If two models make similar predictions in most cases, it will be difficult to recognize one model over the other and we should not

[1]The players max speed is 1m/cycle.

expect any recognition algorithm to get near perfect accuracy. The concept of separability will give us a standard to compare how well the models are being recognized in the real system. While separability seems to be a basic statistical concept, the author is not aware of standard definitions or calculations which fit this problem.

Separability will of course be a function of an *entire set* of models, not a property of any one model. Also, separability must be a function of the number of observations; as we get more information, we expect to be able to identify the correct model more often.

For illustration, consider a simple example where you have two sets of models of coins. In the first set, one model says that heads comes up 99% of the time and the other says heads comes up 1%. For the second set, the models say 51% and 49%. Separability asks the question: if the world is exactly described by one of the models in our set (but we don't know which one), how does the number of observations affect the probability we will identify the correct model? Clearly, we are much more likely to identify the correct model for the first set than the second set because the predictions are so different.

We will develop the concept of separability in four stages. First we will consider the separability of two distributions given one observation, then the separability of two distributions given multiple observations, then the separability of sets of distributions, and finally the separability of a set of models.

Start with two distributions A and B over two dimensions. The question we are interested in is: if we are seeing an observation probabilistically generated from A, what is the probability that the naive Bayes update (starting with a uniform prior) will return with A being the most likely distribution? Equivalently, what is the probability mass of A in the area where the probability distribution function (i.e. pdf) of A is greater than the pdf of B? Of course, we are also interested in the case where B is the distribution generating observations, and in general these probabilities can be different.

Note that the concept of separability we are interested in here is similar to relative entropy or Kullback-Leibler distance [Cover and Thomas, 1991]. The relative entropy of distribution A to distribution B is (where $f_A(x)$ is the pdf of A at x):

$$D(A||B) := \int f_A(x) \log \frac{f_A(x)}{f_B(x)} dx \tag{5.7}$$

The important difference is that Kullback-Leibler distance is considering the ratio of f_A to f_B. In our update, we only care whether f_A or f_B is larger; that is, if the wrong distribution comes out as more likely in our update, we don't care how wrong it is. In other words, our loss function is binary (correct or incorrect) and Kullback-Leibler is targeted for a loss function which is not. More precisely, if $I_{A>B}(x)$ is an indicator function for whether

$f_A(x) > f_B(X)$, then our separability of A from B is

$$\int f_A(x) I_{A>B}(x) dx \tag{5.8}$$

If one considers the models in the context of how they are used, then the appropriate loss function may not be binary as indicated here. The loss function should represent how bad it is to use one model when another one is the true model. As will be shown below, computing the separability of *models* is not trivial, nor is computing the true loss function given the complicated use of the models in Section 6.3.1. Therefore, we will simplify the separability computation by assuming a binary loss function.

Now consider seeing multiple observations instead of a single one. Once again, we are interested in the probability that A will have the highest posterior after the naive Bayes update on all of the observations. This probability is challenging to solve for analytically, but Monte Carlo simulation can estimate it.

Our concept of separability extends naturally to a set of distributions rather than just two distributions. We still want to measure the chance that the correct distribution has the highest probability. We can again use Monte Carlo simulation to estimate the probability that the correct distribution will be recognized for any given number of observations.

Finally, the opponent models are not simple distributions. The models are functions from starting world states, starting states of the opponents, and a planned team action to distributions of the opponents' states. We use an empirical test to estimate separability of *models*. For each observation $o_1, \ldots o_k$ from real game play, we repeatedly generate a series of artificial observations

$$(w, s, a, e^1) \ldots (w, s, a, e^n) \tag{5.9}$$

where

- $w \in \mathcal{S}_W$ is the starting world state
- $s \in \mathcal{S}_O^p$ is the set of opponents' starting states
- $a \in \mathcal{A}$ is the planned team action
- $e \in \mathcal{S}_O^p$ is the set of opponents' ending states
- n is the number of observations for which we want to estimate the probability of correctness

w, s, and a are taken from the real observation o_i. Each set of ending opponents' states $e^1 \ldots e^n$ is sampled from the distributions output by the correct model (the model for which we want to estimate the probability of the naive Bayes being correct). For each sequence of artificial observations, the update is performed. Averaging over all series of observations, we can estimate the probability of a model being correctly recognized, given n observations.

Figure 5.2 shows, for each model, the probability of that model being correctly recognized as a function of the number of observations. One can see that if the models perfectly capture the opponents, after only a few updates, the recognition accuracy should be quite high (more than 85%).

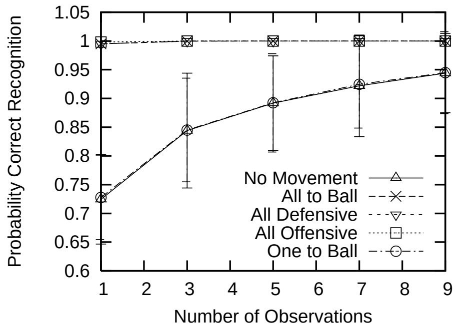


Figure 5.2: Separability of the models for a standard deviation increase rate of 0.3. The "No Movement" and "One to Ball" lines are the lower two lines. Error bars of one standard deviation are shown on only those two lines because the error bars are too small to be visible on the others.

Given the accuracy we would expect if our models were perfect, we can empirically see how well our recognition algorithm actually does. For each model, we programmed teams of agents to act as close as we could manage to the behavior predicted by the model. Note that we can not make this behavior perfect because of the partial observability and dynamics of the world. We then ran our team and coach against each of these programmed teams and recorded all the observations obtained. For each number of observations n, we examine all contiguous sequences of n observations. For each sequence of observations, we perform the naive Bayes update (starting from a uniform prior). For each model, the empirical recognition accuracy is the percentage of the time that the correct model came up as most likely after that series of observations.

Figure 5.3 summarizes the recognition accuracy of the models. For most models we achieve 90% recognition after just 4 observations. The accuracies track the separability from Figure 5.2 with some exceptions. First, there is more confusion among all models for the lowest couple of observations. This suggests that the tails of the distributions are heavier than what the Gaussian model suggests, or possibly that the distributions are multi-modal. Second, the One to Ball model is confused with the other models less than the separability indicates, which probably means that the actual variation for the team is less than that suggested by the model. Finally, the All Offensive model never achieves the near 100% performance suggested by the separability. This result reveals something missing from the model. Namely, during the actual executions, the players on the team will not position themselves offsides. The details of the offsides rule in soccer are not important, but basically it prevents the players from moving too far to the offensive side. The offsides rule, which is ignored by the model, prevents the model's predictions from being completely accurate. Overall, note that the recognition accuracy is quite high after very few observations.

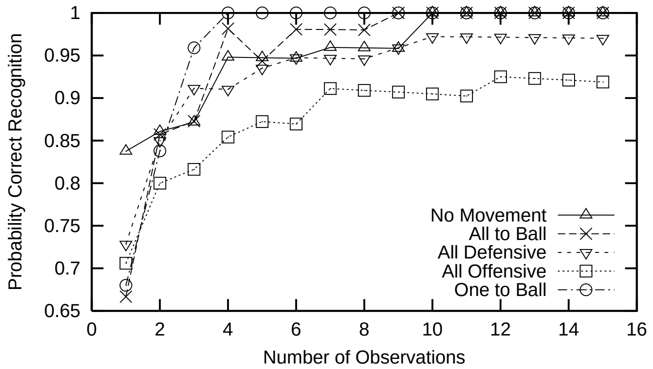


Figure 5.3: Given a number of observations and a model that the opponent is approximately executing, this graph shows the percentage of time a contiguous sequence of real observations results in the correct model being most likely after our naive Bayes update. This graph can be compared to the separability from Figure 5.2, but it should be noted that the axes have different scales.

The exact separability and rate that the accuracy increases will of course change given the set of models that one chooses. However, the observed accuracies tracked the theoretical separability quite well, suggesting that the assumptions made in the model recognition algorithm were good enough. Secondly, the recognition accuracies increased quite quickly, which is at least suggestive that other model sets may have similarly quick recognition.

5.2 Team Models

This section discusses two models of simulated robot soccer teams which describe the overall play of a team: the formation, or general positioning, of a team and the patterns of passing on the field. The learning and use of the models is described, followed by an experimental exploration of their use.

This section is phrased in soccer terminology in many places (such as "formation" and "passing"). However, spatial formations have been important in other robotic behaviors [e.g., Balch and Arkin, 1998] and the formations described here are an instance of the general concept. Similarly, the patterns of passing of a team are an instance of looking for repeated patterns of behavior in the execution trace of an agent team.

5.2.1 Formations by Learning

One important concept in robotic soccer is that of the formation of the team [Stone and Veloso, 1999]. Similarly, teams of robots also maintain particular spatial relationships [e.g., Balch and Arkin, 1998]. The concept of formation used by CLang is embodied in the "home" action. The home areas specify a region of the field in which the agent should generally be. It does *not* require that the agent never leave that area; it is just a general directive.

Our coach represents a formation as an axis aligned rectangle for each player on the team. Rectangles are already in use by many teams (such as CMUnited [Stone et al., 2000]) to represent home areas so we choose this representation for compatibility. From the home areas, agents can also a infer a role in the team, with the common soccer distinctions of defenders, midfielders, and forwards.

Based on observations of a team, our coach learns the formation of that team. The algorithm's input is the set of locations for each player on a team over one or more games. The learning then takes place in two phases.

Phase 1: The goal of the first phase is, for each agent, to find a rectangle which is not too big, yet encompasses the majority of the points of where the agent was during the observed games. The learning is done separately for each agent with no interaction between the data for each agent. First the mean position of the agent (c_x, c_y) is calculated, as well as the standard deviation (s_x, s_y). Second, a random search over possible rectangles is done. The rectangles to evaluate are generated from the following distribution (for the left, right, top, and bottom of the rectangles), where $N(m, \sigma)$ represents a Gaussian with mean m and standard deviation σ (which is a parameter for the search)[2]:

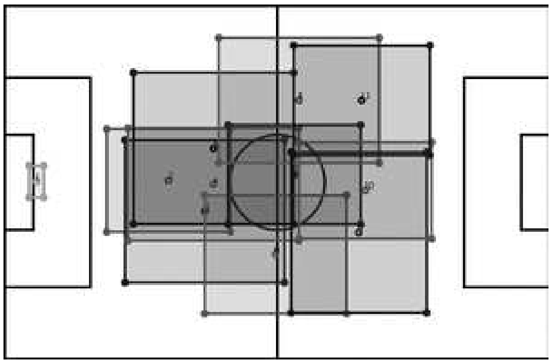
$$(N(c_x - s_x, \sigma), N(c_x + s_x, \sigma), N(c_y - s_y, \sigma), N(c_y + s_y, \sigma)) \tag{5.10}$$

The evaluation function E takes three parameters: γ, β, M. E of rectangle R is then (where A is the area of R and f is the fraction of points inside R):

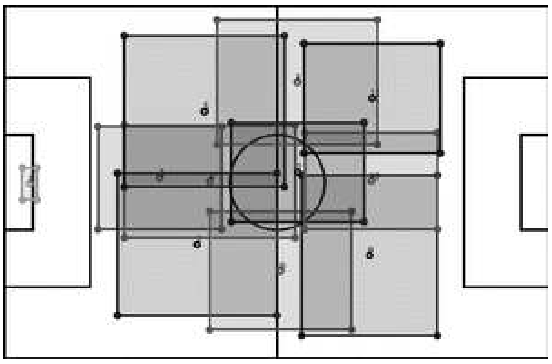
$$E(R) = \gamma f^{\beta} + (1-\gamma)\left(1 - \frac{A}{M}\right) \tag{5.11}$$

All parameters were hand tuned and we used the following values: $\sigma = 10$, $\gamma = 0.95$, $\beta = 1/3$, and $M = 900$.

Phase 2: The first phase of learning ignores correlation among the agents. In fact, it quite common for all agents to shift one direction or another as the ball moves around the field. These correlated movements tend to cause the average positions (and therefore the rectangles from phase 1 of the learning) to converge towards the middle of the field, as shown in Figure 5.4(a). The second phase is designed to capture pairwise correlations among the agents. The rectangles will be moved, but their shape will not be changed.



(a) After Phase 1



(b) After Phase 2

Figure 5.4: The learning of the CMUnited99 formation from RoboCup2000 games. Each individual rectangle represents a home region for an agent, with a dot in the middle of each rectangle. By looking at the location of these center dots, one can see that the home regions in (a) are more clustered in the middle of the field than the home regions in (b).

For this phase, conceptually think of a spring being attached between the centers of the rectangles of every pair of agents. The resting length for that spring is the observed average

[2]We use a coordinate frame where (0,0) is in the upper left.

distance between the agents. Also, attach a spring with a resting length of 0 between the center of a rectangle and its position at the end of phase 1. A hill-climbing search is then done to find a stable position of the system. Figure 5.4(b) shows an example of the effect of the second phase of learning.

Now we describe the details of phase 2. First, the observed average distance t_{ij} between every two agents is calculated. Next, for each pair of agents, a value α_{ij} roughly corresponding the the tension of the spring in the above description is calculated as follows (w, b, and m are parameters):

$$\alpha_{ii} = b * w \tag{5.12}$$
$$\alpha_{ij} = b * e^{mt_{ij}} \quad (i \neq j) \tag{5.13}$$

Here b is the y-intercept of the α function and m is a slope parameter. The idea is to make the shorter springs more tense, and therefore have more impact on the final position of the agent's rectangle. We use this distance weighting because we believe that the correlation of movements of nearby agents is more important than for far away agents. Since $t_{ii} = 0$ for all i, Equation (5.12) (used instead of Equation (5.13)) reduces the impact of the connection to the original position, with w being a parameter which controls that weighting. We used $w = 0.5$, $b = 0.1$ and $m = -0.01$ here.

At each step of the hill-climbing search, a particular agent p is chosen at random to have its rectangle shifted. All other rectangles are held fixed. For all i, let o_i be the original position of rectangle i and let c_i be the vector of the center of current position of rectangle i. The evaluation function is then:

$$\alpha_{pp} \left(\mathrm{dist}(c_p, o_p)\right)^2 + \sum_{j \neq p} \alpha_{pj} \left(\mathrm{dist}(c_p, c_j) - t_{pj}\right)^2 \tag{5.14}$$

The gradient of the evaluation function with respect to c_p is easily calculated and a small step is taken in the direction of the gradient (with learning rate 0.1).

A learned formation can be used in two ways.

Imitation We identify a team which performed well against this particular opponent and imitate that winning team's formation.

Formation Based Marking The coach observes the previous games of the opponent we will play and learns their formation. Each of the defenders is then assigned (greedily, by distance between the centers of the home area rectangles) one of the forwards of the opponent to mark for the whole game. The mark command is a standard CLang action as described in Section A.1.3. This use of the formation assumes that the opponent team will use the same formation in the next game as they used in the observed games.

5.2.2 Rule Learning

The passing patterns of a team are an important component of how the team plays the game. Our coach observes the passes of teams in previous games in order to learn rules which capture some of these passing patterns. These rules can then be used either to imitate a team or to predict the passes an opponent will perform.

The coach observes previous games and for every passing instance gathers the following data:

Passer location The (x, y) location of the passer at the start of the pass

Player locations For all teammates (except for the goalie) and opponents, the distance and angle (in global coordinates) from the passer at the start of the pass

Receiver location The (x, y) location of the receiver at the completion of the pass

In order to get more training instances from each game, we make one further assumption about the behavior patterns of a team. Namely, we assume that if the agent kicking the ball for the pass began at any point along the trajectory of the pass, it would still of performed the same pass.

More generally, this assumption is that for a macro-action that proceeds through a sequence of states, if an agent would have performed the macro action at the beginning of the sequence, the agent would have performed the action at any intermediate state in the sequence. This assumption is especially reasonable when the macro action will accomplish the same goal state (in soccer, the receiving agent having the ball) when the action is performed at any intermediate state.

For soccer, this means that for every real observed passing instance, several imaginary passing instances are created. Every 5 cycles (half a second) along the path of the ball an imaginary pass is created. The locations of all agents are stored as where they were at that time except for the passer, which is moved to at the ball location at that time.

The next step is to cluster the passer and receiver locations. We use Autoclass C [Cheeseman et al., 1988] to create two sets of clusters: one for all passer locations and one for all receiver locations.[3] Some domain-specific tweaking had to be done in order to get useful clusters from Autoclass. Rather than changing the clustering algorithm, we chose to modify the input data. In particular, for every non-imaginary pass's receiver location, we produced multiple input points for Autoclass. There are two parameters m and s to this point replication. Each point had between 0 and m extra points created by a sampling a Gaussian with mean at the original point and a standard deviation of s. We typically used $m = 6$ and $s = 3$. An example of the clusters produced is shown Figure 5.5.

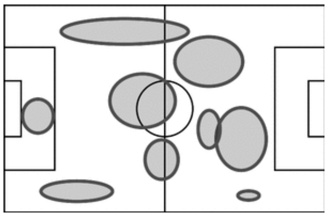


Figure 5.5: Example clusters learned with Autoclass. Each shaded area represents a learned cluster of passer or receiver locations for a pass.

One important caveat must be noted. The clusters returned by Autoclass are not fixed areas of the field. Rather, each cluster is represented by a mean and standard deviation in 2 dimensions. Each data point has a probability of belonging to each cluster. However, several adjustments must be made to allow the learned rules to be expressible in CLang (see Section 2.2.2). First, we ignore the probabilistic membership and assume each data point belongs only to the most likely cluster. Second, the clusters are converted to fixed regions of the field with boundaries at one standard deviation. If CLang were extended to represent Gaussian ellipses, these changes would not be needed and the learning could proceed virtually unchanged.

Once the clustering has been done, our coach learns rules via C4.5 [Quinlan, 1993]. 20% of the data is reserved for testing. The input variables are:

Passer location A discrete value for the cluster (absolute coordinates)

Teammate and opponent locations 2 continuous variables for distance and angle of each agent from the passer (relative to passer coordinates). These do *not* use the learned clusters.

Receiver location A discrete value for the cluster (absolute coordinates)

[3]In Autoclass terms, we used "real location" variables with an error of .0001, which is the observation error for the coach.

The rules from C4.5 are then transformed into rules in CLang, depending upon how they are to be used. If the rules are to be used for imitating the passing of the observed team, the action attached to the condition of the rule is to pass to the area of the field (i.e. the cluster) which the rule predicts. If the rules are to be used to predict the pass of the opponent, the action is to mark that region of the field. We call these rules offensive and defensive, respectively.

In order to make the process more concrete, we now provide an example of an learned offensive rule.[4] A detailed explanation of the meaning of the rule follows in order to help the reader decipher the CLang.

```
1  ((and (play_mode play_on)
2       (bowner our)
3       (bpos "PLINCL0")
4       (ppos our 1 1 {6} ((pt ball) 23 1000 -180 360))
5       (ppos opp 1 1 {10} ((pt ball) 0 1000 151 29)))
6    (do our {2 3 4 5 6 7 8 9 10 11} (bto "PLOUTCL1" {p}))
7    (do our {11} (pos "PLOUTCL1")))
```

Lines 1–5 are the conditions for the rule and lines 6–7 are the directives. Line 2 says that some player on our team is controlling the ball. Line 3 says that that the ball in a particular cluster (“PLINCL0” is the name of the cluster). Lines 4 and 5 are conditions on the position of particular players. Line 4 says that teammate number 6 is at least 23m away, while line 5 says that the angle of opponent number 10 is between 151 and 180 degrees. Line 6 instructs all players on our team (except the goalie who is number 1) to pass the ball to a particular cluster. Line 7 instructs a teammate number 11 (whose home formation position is closest to cluster “PLOUTCL1”) to position itself in that region.

The defensive rules operate similarly. The difference is in the action that the team is advised to do. If the rule predicts that the opponent is about to pass to a particular area, nearby agents are advised to mark the line to the target region (see Section A.1.3).

5.2.3 Experimental Setup and Results

The language CLang (see Section 2.2.2) was adopted as a standard language for a coach competition at RoboCup2001. Four teams competed, providing a unique opportunity to see the effects of a coach designed by one group on the team of another.

[4]The format here is the 7.xx version of CLang, which is slightly different from the 8.xx version described in Section 2.2.2 and Appendix A. However, the differences are minor and the reader should have no trouble making the translation.

The models described above are part of a fully functioning coach agent for the simulated robot soccer environment. We participated in the coach competition, which consisted a single game in each test case. This section reports on our thorough empirical evaluation of our coach and the techniques used, which we performed after the competition. These experiments are intended to show that the models presented learn something about the team modeled and that such models can be useful for giving advice to a team.

Each experimental condition was run for 30 games and the average score difference (as our score minus their score) is reported. Therefore a negative score difference represents losing the game and a positive score difference is winning. All significance values reported are for a two tailed t-test.

We use nine teams for our evaluation as shown in Table 5.1.[5] We will use the nicknames indicated for the remainder of this section. We prepend "C-" to indicate the coach from that team, so for example "C-WE" is the coach from the WrightEagle team. The four teams that understand CLang are WE, HR, DD, and CM. Team descriptions for teams from 2000 can be found in Balch et al. [2001], descriptions for teams from 2001 can be found in Coradeschi and Tadokoro [2002], and CMU99 is described in Stone et al. [2000].

Nickname	Full Name	Year	Institution
WE	WrightEagle	2001	University of Science and Technology, China
HR	HelliRespina	2001	Allameh Helli High, Tehran
DD	DirtyDozen	2001	University of Osnabrück
CM	ChaMeleons	2001	Carnegie Mellon University
GEM	Gemini	2001	Tokyo Institute of Technology
B	Brainstormers	2001	University of Karlsruhe
VW	VirtualWerder	2000	University of Bremen
ATH	ATHumboldt	2000	Humboldt University
FCP	FCPortugal	2000	Universities of Aveiro/Porto
CMU99	CMUnited99	1999	Carnegie Mellon University

Table 5.1: Teams used in opponent modeling experiments. All teams were competitors at the international RoboCup events. Several teams kept the same name over several years of competition, so the year field indicates which public version we used.

In order to measure the performance of our coaching algorithms in this domain, we

[5] In all experiments, we slowed the server down to 3-6 times normal speed so that all agents could run on one machine. This was done for convenience for running the experiments. We tried to verify that agents were not missing cycles and while this setup shouldn't affect outcomes compared to running on several machines, the design of the server makes it impossible to say for sure.

must have a baseline to compare ourselves to. One answer is to compare to the performance of the team with no coach. However, one problem in coaching systems is to understand what magnitude of effect the coach can have. Since the agents can ignore or interpret some advice, it is important to judge to what extent the coach's advice can affect the performance of the team. Therefore, we also compare to a "random" coach. The random coach is a mode of our coach C-CM and we use some components of other techniques to help generate the random advice. The random coach advice is generated as follows:

- A number of conditions c is chosen from a Geometric(0.4)[6] distribution.
- c conditions are chosen at random from conditions actually used for rules in the rule learning (Section 5.2.2) and combined with "and."
- A number of directives is chosen from a Geometric(0.4) distribution (each directive contains a list of agents, a positive/negative advisory mode, and an action)
- For each directive, each agent is included with probability 0.8 and the advisory mode is positive with probability 0.8.
- For each directive, a random action is selected uniformly. If a region of the field is needed as a parameter to that action (such as "pass to"), it is selected uniformly from the clusters used in the rule learning (see Section 5.2.2).

The random coach generates 35 rules at the beginning of the game and is then silent for the rest of the game.

Our first set of experiments replicated the coach competition at RoboCup 2001 with a larger number to games so that we can better account for noise in the outcomes. For the four competitors, each coach was paired with each team. 30 games of each condition were run. For the formation coaching, our coach (C-CM) observed all of Brainstormer's and Gemini's games for the first round of the regular RoboCup team competition. Advice was sent to imitate the Brainstormers formation and formation based marking was used against Gemini's formation. For the rule learning, a single game of Brainstormers against Gemini was observed. Information about the rule learning can be found in Table 5.2. The decision to imitate Brainstormers was made because they performed the best of any of the opponents that Gemini had played up until that point.

The results of this first set of experiments can be seen in Table 5.4. First, for the WE row, none of the differences between the entries are significant ($p > .17$ for all pairs). We hypothesize that WE is effectively ignoring what the coach says since even the random coach has no significant effect on the players.

[6] A Geometric(p) distribution is a discrete distribution where the probability of an outcome $x \in [1, \infty)$ is p^x.

		Autoclass		C4.5	
	# Examples	# Start clusters	# End clusters	# Rules	Accuracy on Test Set
B v GEM	417	11	5	17	75.3%
GEM v B	51	2	9	8	40%
CMU99 v VW	1261	17	9	22	35.1%
VW v CMU99	497	7	4	13	43%
FCP v ATH	1638	8	9	34	61%
ATH v FCP	114	4	5	9	73.9%

Table 5.2: Values for rule learning. The first team listed is the one whose passes are learned. For example, the first row is about Brainstormer's passes against Gemini. The last column represent the accuracy of the learned C4.5 decision tree on the 20% of data reserved for testing.

Team	Score Difference
WE	9.1 [8.1,10.2]
HR	1.6 [1.1,2.1]
DD	-17.2[-18.1,-16.3]
CM	-6.5 [-7.2,-5.9]

Table 5.3: Score differences (positive is winning) against a fixed opponent (GEM) for teams without a coach. The intervals shown are 95% confidence intervals

For the HR row, it is clear that the team is listening to the advice, because C-DD has a highly significant ($p < .000001$) negative effect on the team. However, both the C-HR (the coach designed for that team) and our coach C-CM have no significant effect on the score ($p > .44$). Skipping down to the CM row, all the coaches have a significant positive effect on the team CM ($p < .005$), with our coach C-CM (the coach designed with CM) having the greatest effect.

For the DD row, the notable effect is the large goal change (+8.4) for the team DD using our coach C-CM. Even though the team and the coach were not designed together, our coach can help their team. A more detailed analysis of this effect can be found in the second set of experiments below. For the rest of the C-CM column, our coach helps CM($p < .000001$), and causes no significant effect on the other two teams ($p > .44$). C-CM never hurts a team's performance.

In short, use of our coach can have a significant positive effect on the team coached.

	Coaches			
Team	Random	C-HR	C-DD	C-CM
WE	1.0 [0.1,1.9]	0.2 [-1.0,1.4]	0.7 [-0.2,1.7]	0.0 [-1.0,1.1]
HR	X	0.0 [-0.5,0.5]	-3.2[-4.0,-2.4]	0.2 [-0.2,0.8]
DD	X	X	-1.4 [-2,7,-0.1][7]	8.4 [7.6,9.3]
CM	-8.3 [-9.3,-7.3]	2.1 [1.4,2.8]	1.3 [0.7,1.9]	4.4 [4.0,4.9]

Table 5.4: Score differences (positive is winning) against a fixed opponent (GEM) for four teams and coaches. The differences shown are the change relative to the score differences with no coach shown in Table 5.3. The intervals are 95% confidence intervals. Note that the WrightEagle (WE) coach is not shown because we were unable to run it. Also, an 'X' in a location indicates that we were unable to run those experiments because the agents consistently crashed.

Except for the WrightEagle coach that we were unable to run, our coach had the best performance of the coaches at the coach competition in RoboCup 2001.

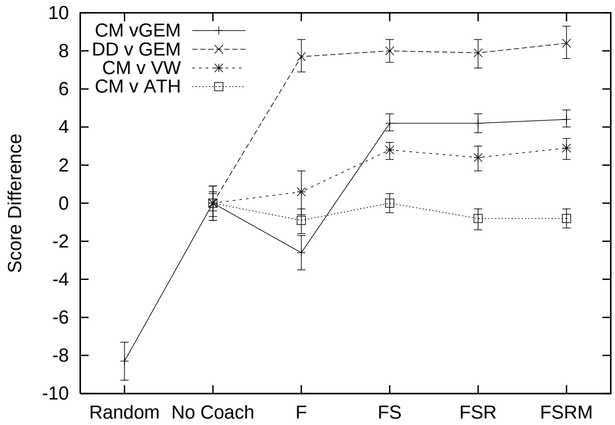
The experiments above for our coach C-CM represent the combination of several different techniques for producing advice. This next set of experiments will serve to identify which techniques provide this positive effect. To do this, we ran a sequence of games with different combinations of the four techniques: formation imitation (F) from Section 5.2.1, set plays (S) from Chapter 6, offensive and defensive rules (R) from Section 5.2.2, and formation based marking (M) from Section 5.2.1.

For playing against VW, our coach (C-CM) observed 5 games of CMU99 playing against VW. Advice was sent to imitate the CMU99 formation and formation based marking was used against VW's formation. Rule learning was also done for those games. Similarly, our coach learned from 10 games of FCP playing against ATH. Information about the rule learning can be found in Table 5.2.

The results from the second set of experiments are shown in Figure 5.6. The CMvATH set is different from the others in several respects. No combination of the techniques resulted in an improvement for CM, and several combinations (F, FSR, FSRM) resulted in significantly worse performance ($p < .05$) compared to no coach.

For the other teams, the combination of all techniques (FSRM) is always significantly better than no coach ($p < .00001$). Looking at the individual techniques is also illustrative. Sending a formation sometimes helps the team (DD v GEM) and sometimes hurts the

[7] One agent on the DD team crashed each time this experiment was run, so it may not be meaningful to compare these results to the others shown here



CMvGEM	DDvGEM	CMvVW	CMvATH
-6.5 [-7.2,-5.9]	-17.2 [-18.1,-16.3]	-2.8 [-3.7,-1.9]	1.2 [0.8,1.7]

Figure 5.6: The score difference of teams coached by a random coach and various techniques of C-CM. The score differences have been additively normalized to the no coach values shown in the lower table. All error bars are 95% confidence intervals. Note that we do not have random coach results for all cases. "F" represents just sending a formation; "FS" represents a formation and setplays (see Chapter 6), "FSR" also includes offensive and defensive rules, and "FSRM" adds formation-based marking.

performance (CM v GEM), even though exactly the same formation is sent in each case. Clearly, the coach needs to learn something about the team being coached.

Except for the CM v ATH line, neither the rules (FSR) nor the formation based marking (FSRM) make a significant impact on the score difference of the games (compared to FS). The formation based marking was a minor part of the coach and it is no great surprise that its impact is small. The rule learning, however, was the most ambitious of the coaching techniques used. The test set accuracies reported in Table 5.2 suggest that the learning is able to capture patterns in the passing of the teams. However, the learning of these patterns is not reflected in the final score of the teams, though it should be noted that the rules do not have the large negative effect that the randomized rules do. It is likely that imitating just the passing pattern reflected in the rules is not enough to help the team significantly. The environment learning described in Chapter 4 covers much more of the environment and has better performance. Using the passing patterns identified to produce more effective advice for the team is an interesting direction of future work.

Overall, the results are mixed. The formation learning worked successfully to produce advice to imitate a good team. The formation based marking, which uses a learned formation to predict opponent strategy, produced no significant effect on the performance of the team. The passing rule learning, while apparently capturing some patterns of a team, failed to produce a significant effect when advice was generated from it. These models are a beginning and an existence proof for opponent models in this complex multi-agent domain.

5.3 Summary

This chapter has covered three different opponent model representations. The first model is a probabilistic prediction of opponent locations based on our team actions. These models were written by hand, but based on online observations, the coach selects the best matching model to use in determining future actions. The second model is a formation, the spatial correlations and patterns of a team. The last model, a team passing model, is a set of learned rules describing patterns of team actions. The formation and passing models are learned from past observations and are used both to imitate successful teams and to predict the actions of the current opponent. All models are empirically evaluated in the simulated robot soccer domain. The experiments show mixed results when using the models in the soccer domain, suggesting both that improvement can be achieved and that further work needs to be done to determine how to best produce advice based on the predictions of the models.

Chapter 6

Coaching Distributed Team Agents

The previous three chapters have considered how a coach can generate and adapt advice. This chapter considers the last of the sub-questions of coaching: "What format does advice take?" In particular, in order for advice to be effective, it must be in a format that can be operationalized by the agents. When a team, rather than a single agent, is using the advice, the problem becomes even more difficult. While in previous chapters the coach was often providing advice to whole teams of agents, the advice was not specifically tailored for the needs of distributed agents. This chapter considers the need for synchronization among agents with different and conflicting views of the world.

6.1 Introduction

A coach agent with periodic communication with the distributed team agents is in a good position to create plans for the agents to execute. Planning is generally easier to do in a centralized fashion. Since the plan must be executed by multiple agents, a natural question to ask is how coordination information can be effectively coded into the plan. We present Multi-Agent Simple Temporal Networks (MASTNs) and an associated distributed plan execution algorithm in answer to this challenge.

We will be using the simulated robot soccer environment as described in Section 2.2. We specifically focus on situations where the game is stopped. At these times, a team can coordinate and generate a plan of action to execute once the ball is back in play. These plans are known as setplays. In the standard rules for the simulated robot soccer environment, this is also one of the times that the coach is permitted to talk. Therefore the coach can make a plan which is then executed by the distributed agents. Several preset setplay plans have been introduced that provide opportunities to position the teammates strategically and have been shown to contribute to the performance of a team [Stone et al., 2000, 1999, Veloso et al., 1998]. We contribute adaptive setplays which are created online by the coach agent. This planning process incorporates a model of the opponent players' movements. These models are described fully in Section 5.1. This chapter describes the plan representation, plan execution, and, in the context of simulated robot soccer, the plan creation.

Our coach agent compiles an overall view of the game and teams in order to provide centralized planning for the distributed agents. Here is a brief overview of our approach:

- The coach agent is equipped with a number of pre-defined opponent models. These models are probabilistic representations of opponents' predicted locations as described in Section 5.1.
- When the game is stopped, the coach takes advantage of the available time to create a team setplay plan that is a function of the modeled opponents' behavior. The plan is generated by a hill-climbing search in plan space. The evaluation function embeds the predictions of the opponent model perceived to be the most likely during the game. The plan generation notably uses the model *to predict* the opponent agents' behaviors. Section 6.3.1 describes this process.
- The plan is encoded in a novel plan representation, a Multi-Agent Simple Temporal Network (MASTN), which is a refinement of a Simple Temporal Network [Dechter et al., 1991]. This representation effectively captures the temporal dependencies between the plan steps, and explicitly records bounds on the expected execution times of the actions. Section 6.2.1 covers the representation and Section 6.3.2 gives information about encoding the plan into an MASTN.
- While the coach is a centralized planning agent, the agents must execute the plan in a fully distributed manner with noisy, incomplete views of the world state. The MASTN plan, as generated and delivered by the coach to the teammates, includes the necessary information for the agents to execute and monitor the team plan in a distributed manner. Section 6.2.3 describes the execution algorithm.
- The coach observes the execution of the plan in order to refine the selection of an opponent model for future plans. Section 5.1 gives the details.

The MASTN representation and execution algorithm are explicitly put forth in non-soccer specific terms. The plan generation algorithm uses an evaluation function based on specific soccer knowledge, but the general hill-climbing strategy should be applicable elsewhere.

A coach using these techniques was part of a team that competed in the RoboCup competitions in 2000, 2001, 2003, and 2004. The coach created a variety of setplay plans, adaptively responding to completely unknown opponent teams. Unfortunately, the design of the coach competition has prevented any meaningful scientific results being drawn from competition results as the noise in the games has, in most cases, been significantly larger than differences between the coaches.[1] Instead, we present controlled empirical results explaining and demonstrating the effectiveness of this approach.

6.2 Multi-Agent Plans

Our coach agent generates movement plans for its teammates and the ball. The coach is a centralized planner, but the execution must be done in a fully distributed fashion. Therefore, the coach must encode sufficient information into the plan to allow the agents to coordinate and identify failures during execution.

The domain-independent portions of the plan representation and execution process are described in this section, as well as their use in the simulated robot soccer environment.

In order to make the simulated robot soccer discussions more clear, we first illustrate what an execution of a setplay plan looks like. Figure 6.1 shows the movements of the ball and players over time. Teammate 1 starts with possession of the ball. The setplay consists of teammate 1 passing to teammate 2, which moves to receive it. Simultaneously, teammate 3 moves forward. Teammate 2 then passes to teammate 3. Note that there are agent actions going on in parallel and that some actions (like the passes) require coordinated efforts of more than one agent. The plan in Figure 6.1 will be used to illustrate the representation and execution algorithm in the following sections.

6.2.1 Plan Representation: MASTN

We introduce Multi-Agent Simple Temporal Networks (MASTNs) as a plan representation for distributed execution. MASTNs are a refinement of Simple Temporal Networks as introduced by Dechter et al. [1991]. The refinements allow us to the define the execution

[1]Our coach placed first in 2001 and the experiments in Section 5.2.3 provide the demonstration that this is meaningful. See Kuhlmann et al. [2004] for a similar set of experiments for 2003.

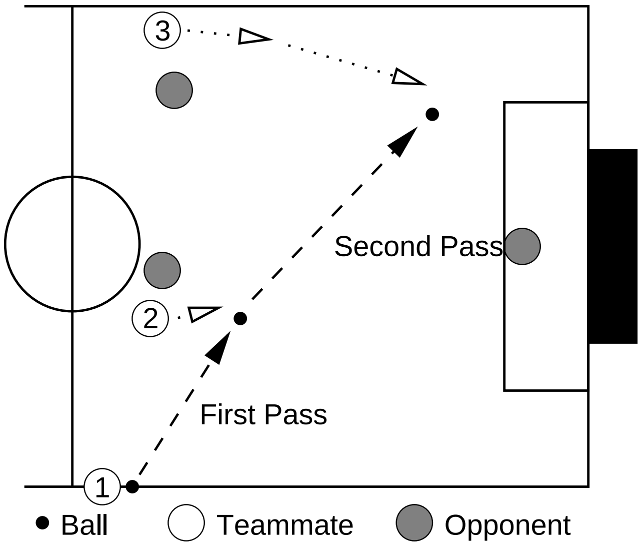


Figure 6.1: An example plan. The arrows indicate movement of the ball or a player.

algorithm discussed in Section 6.2.3. Taken together, the representation and execution algorithm are a significant scientific contribution.

We will first present Simple Temporal Networks, then introduce our refinements to make Multi-Agent Simple Temporal Networks. Our formal presentation is a combination of the notations of Dechter et al. [1991] and Morris and Muscettola [2000].

A Simple Temporal Network is a tuple $\langle \mathcal{V}, \mathcal{E}, l \rangle$ where

$\mathcal{V}$ is a set of nodes. Each node represents a event which occurs at a point in time. Activities with duration must be broken up into two events, one representing the beginning and one representing the end of the activity.

$\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of directed edges between the nodes. Each edge represents a temporal constraint between nodes.

l: $\mathcal{E} \to (\mathbb{R} \cup -\infty) \times (\mathbb{R} \cup \infty)$ is the labeling function on the edges. The label represents a temporal constraint between the events. For example, if $\langle a, b \rangle \in \mathcal{E}$ and $l(a, b) = \langle x, y \rangle$, then event b is constrained to occur at least x time units after a, but no more than y. Note that x and y could be negative (indicating that b should occur before a) or negative or positive infinity (respectively).

STNs are a representation of a subclass of temporal constraint satisfaction problems. A user of an STN often wants to answer questions like "Is there any set of times at which my events can occur such that no constraints are violated?" or "What time should each event be executed so that no constraint is violated?" Unlike more general temporal constraint satisfaction problems, these questions can be answered in polynomial time for STNs.

We can now define Multi-Agent Simple Temporal Networks (MASTNs). We assume we are given the following:

$\mathcal{A}$ A set of identifiers for the agents.

$\mathcal{T}$ A set of node types. Nodes represent events which are brought about by the agents, so this set is similar to the set of actions available in traditional planning problems. The nodes can be parameterized. For example, in the classic blocks world there would be node types for "PickUp(RedBlock)", "PickUp(GreenBlock)", etc. Note that which agent performs an event is explicitly *not* part of a node type as that will be represented elsewhere.

We then define the set of all possible nodes (i.e. events) $\mathcal{N}$ as[2] $\mathcal{T} \times \mathcal{P}(\mathcal{A}) \times \mathbb{N}^*$. The $\mathcal{P}(\mathcal{A})$ represents the set of agents responsible for bringing about this event. The $\mathbb{N}^*$ element is used to provide "pointers" to other nodes (described below). The pointers allow some information to be specified in only one plan node rather than being duplicated.

An MASTN plan is then a tuple $\langle \mathcal{V}, \mathcal{E}, l, O \rangle$. These elements are:

$\mathcal{V} \subseteq \mathcal{N} = \mathcal{T} \times \mathcal{P}(\mathcal{A}) \times \mathbb{N}^*$ The set of nodes of the plan. Note that each node contains more information than in a normal STN which considers each node a black box.

$\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ The set of edges, just as in an STN.

$l\colon\ \mathcal{E} \to (\mathbb{R} \cup -\infty) \times (\mathbb{R} \cup \infty)$ The labeling function on the edges, with the same meaning as in an STN.

$O\colon\ \mathcal{V} \to \mathbb{N}$ An ordering function on the nodes. In other words, for $v_1, v_2 \in \mathcal{V}$, $o(v_1) < o(v_2)$ means that v_1 comes before v_2. This ordering function is used as the address space of the node pointers and to allow the agent to break ties when the temporal constraints allow multiple events to be executed. The details of this tie-breaking are discussed in Section 6.2.3. O is not a full temporal ordering of the nodes.

The authors are not aware of any other work where STNs are used as a basis for a multi-agent plan representation for distributed execution. In particular, by "distributed

[2] Remember that $\mathcal{P}$ is the power set operator and $\mathbb{N}$ is the natural numbers.

execution" we mean that each agent maintains its own perception of the current state of plan; there is no assumption of shared global sensations or plan state. Temporal networks have several properties which are useful in a multi-agent context:

- The network represents the parallelism of agents' actions. The temporal constraints express some basic needed coordination between the agents.
- Temporal constraints can be used to help agents detect failures in the plan. Constraints can naturally catch limits on the extent of events or if an agent fails to take appropriate action at the correct time.
- If an event e is not ready to execute because of temporal constraints, the network represents which event(s) are preventing e from being ready. If an agent is responsible for executing an event that is not ready, the agent can determine where in the world to look to observe the future execution of the event which is preventing e from being ready.

We introduce MASTNs as a way to take advantage of these properties for multi-agent plan execution.

6.2.2 MASTNs in Robot Soccer

This section demonstrates how the MASTN representation is applied to simulated robot soccer. We have to instantiate the domain specific values $\mathcal{A}$ and $\mathcal{T}$. Let $\mathcal{L}$ be the set of locations on the field.

- $\mathcal{A}$ is the set of numbers 1 through 11. Each agent is identified with a unique number.
- $\mathcal{T}$ is the set of node types, some of which have parameters. For example, if a node type t has a parameter of type $\mathcal{L}$, then $\forall x \in \mathcal{L}, \langle t, x\rangle \in \mathcal{T}$. In particular, this means that all parameters are bound and fixed for a plan; the values of parameters to nodes do not change during plan execution.

  Initial Position This node represents the event of all agents arriving at their initial position for the player and the play starting. This node has a parameter which is an element of the set of partial functions from $\mathcal{A} \to \mathcal{L}$, mapping agents to their initial locations. This node is also the root node for execution, called "the beginning of the world" by Dechter et al. [1991].

Start Goto This node represents the beginning of the activity of a given agent going to a location on the field. This node takes a parameter (which is an element of $\mathcal{L}$) indicating where to go.

End Goto This node represents the conclusion of the move begun by the "Start Goto". The node pointer element is used to specify the associated Start Goto node.

Start Pass This node represents the beginning of a pass activity, where one agent kicks the ball to a location on the field. This node takes a parameter which is an element of $\mathcal{L}$ indicating where to pass. While another agent should receive the pass, that information is not explicitly represented in this node.

End Pass A pass is represented with three nodes: a Start Pass (for the kicker to start the ball), a Start Goto (for an agent to start moving to receive the pass), and an End Pass which represents the conclusion of both of those activities. The node pointers are used to specify the related nodes.

Clear Ball This node represents an agent kicking the ball to a location. This node takes a parameter which is an element of $\mathcal{L}$ indicating where to kick the ball. This differs from a "Start Pass" node because no particular agent is expected to get the ball. There is no associated end node, because the plan is always complete after a "Clear Ball" node executes.

An example plan is graphically shown in Figure 6.2. A successful execution of this plan is depicted in Figure 6.1. The top three nodes represent the first pass from player 1 to player 2. The bottom two nodes represent the simultaneous movement of player 3. The last three nodes (in the middle right of the figure) represent the pass from player 2 to player 3.

6.2.3 Plan Execution

Plan execution is done in a fully distributed fashion. This means that each agent maintains its own data structures describing the state of the plan and independently decides what actions to take. However, it is assumed that every agent can get some knowledge of the events in the plan being executed (either by communication or observation), though it is *not* required that agents agree exactly on the timing and sequencing of events. In domains with limited observability or latency in the communication between agents, exact agreement is difficult to obtain. Therefore, flexibility in agent agreement is important. In the simulated soccer environment, the agents do have a shared global clock. In general, however, the

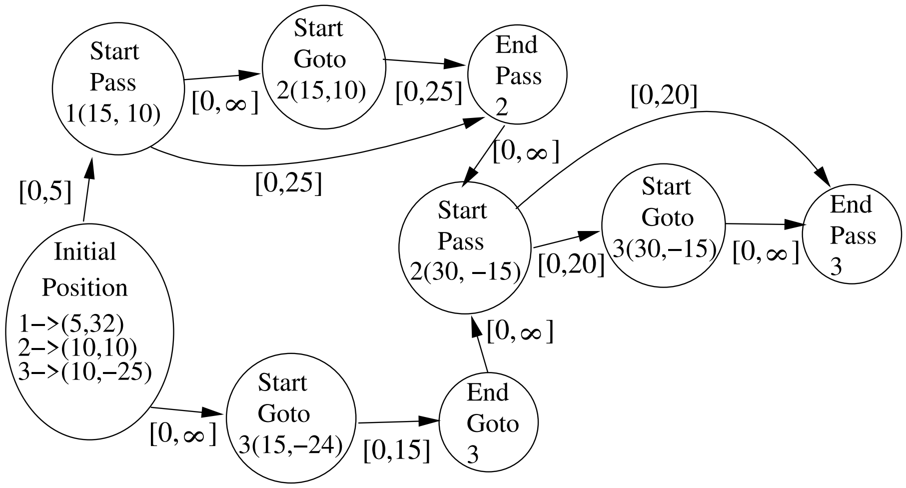


Figure 6.2: An example MASTN. The nodes represent events and the edges represent temporal constraints. The agent responsible for bringing about the occurrence of the event is shown inside each node. The numbers in parentheses are vectors representing locations on the field (elements of $\mathcal{L}$). These vectors are the parameters to the various node types. The node pointers are *not* explicitly represented in this picture. This plan corresponds to the execution depicted in Figure 6.1.

agents do not have to agree on the exact time as long as each one can measure the progress of time accurately. In other words, as long as each agent can measure how long a second is accurately, it does not matter if all their watches agree.

Throughout, we will talk about "executing" a node. This means that the agent is taking individual actions to accomplish the event represented by the node. As discussed in Section 2.2, an agent is not able to precisely control when an event will occur, even when it is the one executing the node. Further, we assume that each agent can only execute one node at a time. That is, all the parallelism in the plan takes place by multiple agents performing actions simultaneously.

In addition to executing various behaviors, the agents will be monitoring the execution of actions that other agents are supposed to perform. If an agent is slow or failing to execute its action, any agents which are temporally constrained by that undone action will wait for it to be done. Note that currently the agents do *not* take over actions if the currently assigned agent fails to execute it.

Muscettola et al. [1998] have described a "dispatching execution" algorithm for STN execution. This method allows easy and efficient propagation of temporal constraints through the network at the cost of adding edges. The first step is to construct the all-pairs closure of the STN (i.e. make an edge from every node a to every other node b whose length is the shortest path from a to b). Muscettola et al. describe a method to then prune some of those edges to reduce the time requirements of plan execution, which is important for STNs consisting of thousands of nodes. However, since we are working with networks of tens of nodes instead of thousands, we do not prune any edges. We then use the dispatching execution algorithms as subroutines to propagate temporal constraints and identify violations.

Table 6.1 shows the full execution algorithm. The algorithm calls some domain specific functions whose purpose will be discussed as the algorithm is covered in detail.

The "initialize" function initializes the data structures for the given plan. In particular, the temporal constraints are set up for the dispatching execution algorithm. Then, at each time step, each agent should run the "execute" function. The execution will result in some action to take for this time step. At the next time step, the "execute" function is run again. Over time, an agent will work to execute various nodes (by taking one of more actions to accomplish each one) in the plan; it may take more than one action/time step to complete execution of a given node. During a successful execution, nodes in the plan will be marked as completed as the plan progresses.

First in the plan execution function is a call to *global_monitor*. This domain specific function should implement monitoring conditions which apply to the whole plan. Returning "fail" indicates that the plan should be aborted. In the soccer environment, we have global monitoring conditions to catch situations like the ball going out of bounds or the opponent intercepting the ball.

Next, in lines 9–11, the nodes in the plan which have not been marked as executed are checked. The domain specific function *node-completed* must be provided to identify when a node has been executed. This decision can be based on perceptions or communication. In general, this decision will depend on the type of node $\mathcal{T}$ and the agents tasked with executing the node.

The functions propagate-constraints and constraints-violated are provided by the dispatching execution algorithm [Muscettola et al., 1998]. The function propagate-constraints will update the "window" data structure with allowable times for the nodes to execute and constraints-violated will indicate whether any temporal constraints have been violated.

The next section of the execute function (lines 16–21) identifies the next node for to agent to execute. The ordering function is used to establish which of the nodes in the plan this particular agent should be working on this time step.

Once the next node to execute is found, preconditions of that node can be checked (lines

```
initialize(p: plan)
    ∀n ∈ p.𝒱
        exectime(n) = ∅
        window(n) = ⟨−∞,∞⟩
        construct all-pairs network for p
execute(p: plan, t: time, a: agent)
    if global-monitor() = fail
        return abort
    ∀n ∈ p.𝒱
        if (exectime(n) = ∅ and node-completed(n))
            exectime(n) = t
    propagate-constraints()
    ∀n ∈ p.𝒱
        if (constraints-violated)
            return abort
    // ℱ is the set of nodes to still execute for this agent
    ℱ = {n ∈ p.𝒱 | exectime(n) = ∅ and a ∈ n.agents}
    if (ℱ = ∅)
        return plan-completed
    // mynode is the next node for this agent to execute
    mynode = argmin_{n∈ℱ} p.O(n)
    if (not node-precondition(mynode))
        return abort
    if (t < window(mynode)[1])
        holdingnodes = {n ∈ 𝒱 | (exectime(n)=∅ and ⟨n, mynode⟩ ∈ p.ℰ
                                          and p.l(n, mynode)[1] ≥ 0 }
        if (holdingnodes = ∅)
            return in-progress // agent needs to wait for time to pass
        else
            look-at(holdingnodes)
            return in-progress
    execute-node(mynode)
    return in-progress
```

Table 6.1: Plan execution algorithm for an individual agent. "exectime" and "window" are data structures maintained during execution. Functions *in italics* must be provided by the user of this algorithm.

22–23). For example, in the soccer environment, a Start Pass node requires that the agent believes the teammate intended to receive the ball will be able to get it. This decision is based on a learned decision tree [Stone, 2000] or other analytic methods [McAllester and Stone, 2001].

If the temporal constraints do not allow the current node to execute (line 24), then there are two cases. If there are no unexecuted nodes which must execute before this one, then the agent just waits for time to pass (lines 27–28). If there is such a node, the domain specific *look-at* function is called to tell the agent to watch for the execution of that node (lines 29–31). In the soccer environment, this is done by having the agent face the point where the execution of that node should occur. In general, this could be done with a communication request or any other observational means.

Otherwise, the agent works towards executing the next node (lines 32–33). Note that the execution of a node may take more than one step and the agent is not required to precisely control when the node executes. These criteria allow more freedom in how the execution of nodes is carried out. For the soccer environment, *execute-node* is written using the reactive CMUnited99 [Stone et al., 2000] layer to get robust performance of such commands as "get the ball" or "kick the ball hard in direction x."

That is the complete algorithm. This algorithm is being run by each agent in parallel every step. Each agent maintains its own perception of the state of the plan; there is no centralized control instructing agents when to perform actions. While STNs used in scheduling tasks can provide coordination of agents, the agents usually have access to shared global state or a shared controller, neither of which we have here. We allow the agents' perceptions of the execution state to differ as long as the difference does not cause a violation of the temporal constraints.

For example, since the agents in the soccer domain use noisy, limited observations to determine when a pass has started, the agents will in general not agree on the exact time that the pass started. Noise can make the ball appear to move when it hasn't or the agent may not be looking at the ball when the pass starts. Even if none of the agents agree on the exact time, the plan execution may still be successful as long as the temporal constraints are not violated.

It is difficult to make precise statements about how much the agents' perceptions of the world are allowed to differ. The difference allowed will depend on the how much flexibility there is in the plan. Plans could be constructed such that there is exactly one time at which each event could execute, giving no flexibility in agents' perceptions. The plans generated here do allow for differences in agent perception and it is an open question how much flexibility is allowed in "typical" plans for the soccer domain or for other interesting domains.

6.2.4 Plan Execution Example

We will now illustrate an execution of the plan shown in Figures 6.1 and 6.2. This section will not attempt to cover all the steps of the algorithm described in Section 6.2.3. Rather, some of the important points relating to distributed execution will be discussed.

Figure 6.3 shows each agent's perceptions of when the events in the plan occur. The shaded events are events for which that agent is responsible. Each agent is responsible for its position in the Initial Position event. After that, agent 3 starts going to its next position. Simultaneously, agent 1 starts the pass to agent 2 (SP1 $\rightarrow$ 2) and agent 1's role in the plan is complete. As shown by the shading, agent 1 no longer tracks the execution of the plan. Some time later, agent 2 realizes that the pass has begun and starts to go to the reception point for the pass (SG2). Meanwhile, agent 3 completes going to the intermediate point (EG3). Once the first pass is complete (EP1 $\rightarrow$ 2), agent 2 passes to agent 3 (SP2 $\rightarrow$ 3). Agent 2's role is then complete. Agent 3 then proceeds to get the ball to complete the pass (SG3 and EP2 $\rightarrow$ 3).

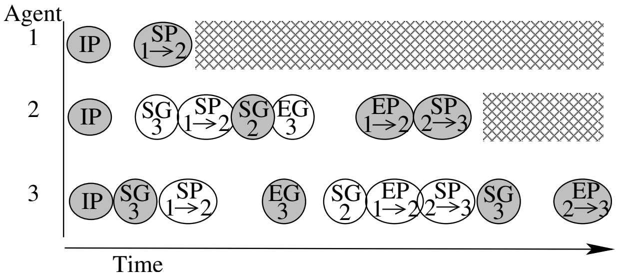


Figure 6.3: Example execution of a plan. This is an execution of the plan in Figure 6.2 and illustrated in Figures 6.1. Each agent (shown on the rows) has its own perception of when events occur; the time an agent perceives an event to occur is shown by the event's horizontal location. The shaded events in each row are the events which that agent is responsible for bringing about. IP stands for Initial Position, SP for Start Pass, EP for End Pass, SG for Start Goto, and EG for End Goto.

Note that the agents perceive events' execution times to be different and that they do not even always agree on the ordering (e.g. SG2 and EG3). However, each agent's perception of the order of events must obey the temporal constraints in the STN (Figure 6.2). Also, at every point, each agent will verify global and local constraints and abort the plan if the verification fails. For example, if an opponent agent intercepts the first pass, the agents will stop the plan, communicating their perception of the need for termination.

6.3 Plan Creation

Given the MASTN plan representation described in Section 6.2, it is still a significant challenge to generate these plans, especially accounting for the predicted behavior of the adversary. We divide the process of plan creation into four steps. The particular implementations of these steps rely on soccer specific knowledge, but this general breakdown would likely be useful in other domains.

1. **Waypoint Planning**: This module plans the ball's trajectory as a series of straight line segments (which are passes or clears in robot soccer). Constraints are put on the length of the segments based upon maximum agent and ball speeds, but agent movements are not part of this step. A model that predicts the opponents' movements is used to help find a good plan.

2. **Role Making**: Given the planned trajectory for the ball, this step creates roles for agents.

3. **MASTN Compilation**: This step takes the output from the role making step and turns it into an MASTN.

4. **Agent Instantiation**: This step performs the role allocation of assigning particular agents to roles in the plan.

Section 6.3.1 describes the waypoint planning and the other three steps are described in Section 6.3.2.

6.3.1 Waypoint Planning

In order to plan the waypoints, we use models of opponent movement. Section 5.1 fully describes these models, but for the purposes of this section, we can just that say that waypoint planning is a path planning problem with straight-line segments and dynamic, probabilistic obstacles (the obstacles are the opponents). Unlike many path planning problems, the obstacles are not fixed regions in known locations. Rather, we have a probability distribution over each obstacle's locations over time.

The opponent models which describe the movement of the obstacles take into account the current positions of the opponents and the predicted actions (i.e. the waypoints) to produce the predicted movements of the opponents. The exact positions of our teammates will be determined from the waypoints, but those positions are not *explicitly* part of the opponent model. Further, the waypoint planning ignores the current positions of the teammates since it is assumed that the team can move into the starting positions before the plan begins.

The problem addressed here is significantly different from a traditional shortest-path planning problem. Selecting a path in this environment inherently involves tradeoffs. There may be one path that is extremely long and goes through an obstacle with very low probability, and another path that is much shorter but has a higher probability of going through an obstacle. To decide which path is better requires a tradeoff in the length and safety of the path. Further, we do not have a single goal position, but rather a ranking of the possible positions. That is, not every setplay will result in a goal being scored, but some final positions from the setplay are better than others.

These constraints make it difficult to apply many of the traditional path planning methods, such as those described in Latombe [1991]. Planning methods that deal with uncertainty do not usually handle obstacles whose location is only probabilistic. Rather, they are more focused on dealing with noisy execution when following the path, or expect replanning to be available. Approaches that deal with moving obstacles do not address uncertainty in obstacle location.

The D^* algorithm, developed by Stentz [1994], was also considered. However, D^* is mostly useful for replanning when obstacles are observed to move, not handling the upfront probabilistic movements we model here. We wanted our coach to come up with a complete plan, not rely on the distributed, executing agents to replan. Replanning would be difficult in this case both because of the partial observability of the executing agents and the unreliable communication.

In order to plan in this challenging domain, we decided to directly specify an evaluation function for paths and use hillclimbing on a set of paths to find a locally optimal path. The evaluation function for the paths will include the processing of the probabilistic opponent model.

Our evaluation function will meet the basic requirement of hillclimbing that nearby paths have similar evaluations. Also, our plan space is about 10^{18} so we can not cover a large proportion of the space in the few seconds available for planning.[3] Note that the

[3]The plan space size was estimated as follows. The field was discretized to 1m. Passes were considered between 8m and 38m and sends between 38m and 55m. A plan could be up to four segments with the last

coach can not plan until the ball actually goes out of bounds because the plan evaluation depends on the current location of the ball and opponents.

We first describe the hillclimbing algorithm and then describe the evaluation function. Our path planning algorithm is shown in Table 6.2. Note that sometimes (decided by the variable A) we move only a single point in a hillclimbing step and sometimes we move the entire tails of paths. By varying the neighborhood considered, the hillclimbing should be able to escape more local minima.

```
S := Set of starting paths
while (there is time left)
    Uniformly randomly remove a path p from S
    Uniformly randomly pick a point x on p
    Uniformly randomly set A to true or false
    bestp = p
    ∀ small displacement vector v
        Make path p' by moving x by v
        If (A)
            In p', move all points after x by v
        If eval(p') > eval(bestp)
            bestp = p'
    Insert bestp into S
    If (time left < half of original time)
        Remove all but current best path from S
```

Table 6.2: Hillclimbing algorithm for waypoint planning

Note that the hillclimbing runs for a fixed amount of time. Halfway through that time the set S is reduced to just the path with the current best evaluation. This allows the second half of the hillclimbing to focus on improving a single path as much as possible.

The set of starting paths are preset and depend on the location of the ball. This dependence is necessary because different types of plans make sense for different situations such as whether the ball is in the middle or side of the field, whether the ball is near our goal, whether this is a corner kick, etc. Since only a small part of the plan space can be explored by hillclimbing, setting good planning seeds can greatly help in finding a high-quality plan. Figure 6.4 gives an example of the hillclimbing seeds for one setplay situation.

---

possibly being a clear.

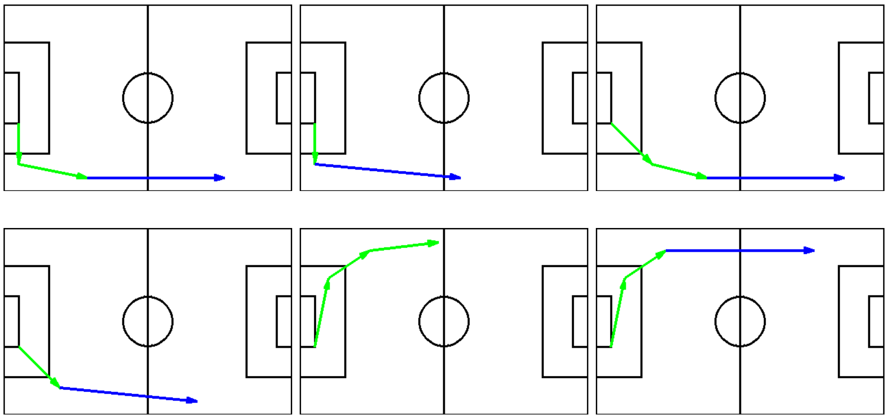


Figure 6.4: Examples hillclimbing seeds. These are the seeds for a goal kick from the bottom side. The longer, darker arrows are clears and the shorter, lighter ones are passes.

The set of starting paths is an easy point for inserting domain knowledge into the system. In particular, for the soccer environment we can give the basic shapes of paths, such as passing to the outside then clearing from a free kick. If there is no domain expert to provide this knowledge, random starting paths or paths taken from past executions could be used.

The crucial part of hillclimbing is the evaluation function (*eval* in Table 6.2). While the particular evaluation function chosen here is specific to the soccer domain, the general idea of hillclimbing in plan space with a domain specific evaluation is applicable to other domains. We use the following weighted factors:

- Player control at end

  If the last segment of the path is a pass, we are in control of the ball at the end of the play (this has value 1.0). If the last segment of the plan is a clear (kicking the ball down field with no specific agent designated to get the ball), this has value 0.0. That is, it is better that our team ends up with control of the ball rather than just kicking it down the field.

- Ball's end location

  The value of the ball's end location also depends on whether we are in control of the ball at the end of the play. In other words, the value of a position of the ball varies based on whether we have a teammate in control of it. Getting the ball near the goal and the opponent's baseline has high value, and just getting the ball further down field is also of high value. The functions are shown graphically in Figure 6.5. The functions for a pass and a clear are almost identical except for two things. First, the clear figure has less value inside the penalty box (if the ball is kicked into the penalty box with no one nearby, the goalie will just grab it). Second, the clear has additional value near the top of the penalty box because it may induce the goalie to move out of the goal, giving a good shot if the ball can be retrieved and passed to an agent on one of the sides. Readers interested in the exact definition of the evaluation function should see the online thesis resources (see Appendix C for details).

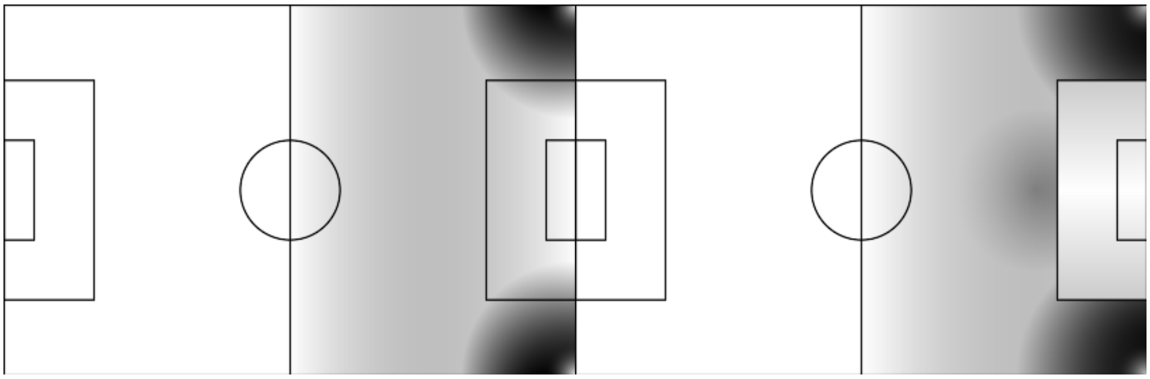


Figure 6.5: Evaluation function for the final location of the ball. Darker is a higher evaluation. The left figure is for a pass (where we control the ball) and the right is for a clear (where the ball is kicked to no particular agent at the end of the play).

- Length of plan

  Since every action has some probability of failure, long plans have a lower chance of succeeding then shorter ones. However, short plans add less to the team behavior simply because they have less time to affect the behavior. This factor makes this tradeoff explicit. Therefore, plans with length 3 (i.e. 3 passes or 2 passes and a clear) have the highest value and the value degrades from there. The values here are heuristic and chosen based on experience with this simulated robot soccer domain. The exact values are shown in Table 6.3.

Length	1	2	3	4	5
Value	0.4	0.8	1.0	0.5	0.2

Table 6.3: Evaluation values for the "length of path" component

- Average path safety and minimum path safety

  These are two different measures of the safety of the path. The safety of each segment in the path is first evaluated and then the average and minimum of those values is computed. A high value here represents greater safety.

  We have two possible ways to compute the safety of a segment. The first method samples the probability distributions of all the players in a triangle from the start of the pass to the end of the pass (see Figure 6.6). Points are sampled at the rate of $\frac{3}{4}$ of a sample per 1 square meter. The width of the triangle at the end of the trajectory is the same as the length of the trajectory. In order to deal with the problem that all probabilities tend to decrease as the length of the pass increases, we multiply the probability by a factor which grows as the time grows. We call this new quantity the "occupancy" of the triangle. If t is the time in cycles (each cycle is 100ms) and p is the probability, we calculate occupancy by the empirically hand tuned equation:

$$1 - (1 - p)^{\frac{t}{6}} \tag{6.1}$$

  The average and minimum values here are just 1.0 minus the average/maximum occupancy of all the passes.

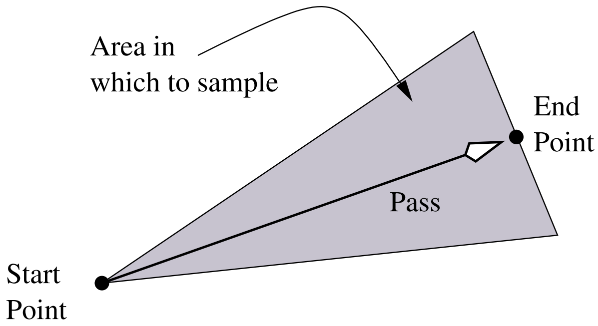


Figure 6.6: Sampling the probability distributions for pass safety

  The second method uses just the means of the distributions of the opponents to estimate the probability of a pass's success. This method [McAllester and Stone, 2001] is used during normal game play to evaluate many different passing options. It was designed to be run extremely quickly, improving the speed of the hillclimbing steps. However, distributional information (such as the variance) is ignored by looking only at the mean of the distributions. The average safety is the average pass success probability while the minimum value is the minimum pass probability. The second method is much faster and was therefore generally, though not always, used for planning. We have not done a thorough empirical comparison of the two methods.

The factors are then added together with the weights shown in Table 6.4. The weight were obtained through hand tuning after the system was implemented.

Factor	Weight
Player control at end	0.22
Ball's end location	0.2
Length of path	0.1
Average path safety	0.33
Minimum path safety	0.33

Table 6.4: Weights for combining the factors for the hillclimbing evaluation function

Hillclimbing has a good anytime characteristic in that as soon as our time for planning is up, we have a plan ready to return (namely the best one so far). It also allows easy application of domain knowledge by giving intelligent seeds. Unfortunately, hillclimbing is also somewhat time consuming and likely will not return the optimal answer, either because of lack of time or a local maximum.

6.3.2 Waypoints to Complete Plan

Given a target path for the ball, the coach constructs an MASTN in three phases:

Role Making A separate role in the plan is created for executing each needed pass (i.e. each intersection of straight line segments). A role consists of all of the locations which the agent will need to move to and where it will kick the ball. This process creates the set of nodes $\mathcal{V}$ and the ordering function O. In particular, nodes of the correct types and their associated parameters and pointers are created. The agent values (members of $\mathcal{A}$) are temporary, to be replaced by the correct agents later in the process. Some domain specific requirements for the soccer environment, in particular for the offsides rule[4], are handled here.

[4]The offsides rule in soccer (and modeled in the Soccer Server) means that a player on one team can not be closer to the opponent goal than the last defender *when the ball is kicked*. For the planning, the offsides rule means that the agents must be aware of when a pass starts in order to stay onsides correctly.

MASTN Compilation The step adds the edges ($\mathcal{E}$) and their temporal constraint labels (l). Domain specific knowledge such as the speed of running and kicking and their normal variations is used to establish the time bounds for execution between the various events.

Agent Instantiation This step assigns specific agents to the roles in the plan. Role allocation is an important problem in multi-agent systems [e.g., Weiss, 1999]. However, for this system, we use a simple domain specific algorithm. The assignment is done using the current formation of the team and a greedy matching algorithm between the agents' home positions and the plan's starting positions.

If the coach knows the current formation, the coach can perform this step. However, this can also be done by the players, as long as their formation information is consistent. For the players, formation information and consistency is obtained through the Locker Room Agreement [Stone and Veloso, 1999]. The Locker Room Agreement is a set of preset knowledge that allows the agents to agree on some aspects of play and strategy changes based on any shared state features (e.g. the game time).

6.4 Empirical Results

This section presents experiments exploring the effectiveness of the planning approach. For the experiments, we will be using a set of predefined opponent models. The details of these models are not especially important for these experiments, but their descriptions can be found in Section 5.1.4.

The complete approach described above has been implemented. The overall goal of the experiments is to test the effectiveness of the various parts, including the use of opponent models and the improvement of a team using the entire approach.

6.4.1 Models' Effect on Planning

One natural question about the planning process is what effect the opponent models have. If the opponent models are meaningful, then the resulting plans should be different for the different models. It is somewhat tricky to isolate just this one effect from the rest of the system. In order to evaluate the differences in the plans produced using opponent models, we compare paths by looking at the area between the paths. For example, in Figure 6.7, the shaded area is the area between the solid and dotted paths. We use this area because it expresses in simple terms how different two paths are, and consequently, how different two plans are. Therefore, the median area difference[5] between a set of plans expresses roughly how much variation there is in a set of plans. The exact numbers are not especially meaningful, but are useful as comparisons of different sets of plans.

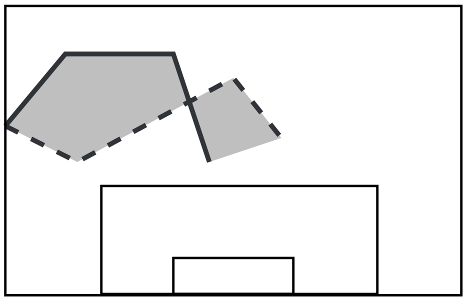


Figure 6.7: Example of the area between paths. The two paths are the solid line and the dotted line and the shaded area is the area between them.

First we look at the variation in our planning seeds. The planning seeds are designed to be far apart in the space of possible plans, so the variation gives some idea of the maximum range we could expect the plans generated by the system to vary. As shown in Table 6.5, the median area difference is 315. Then, we compare the plans generated when the only variation is which opponent model is used for planning. Using a different opponent model for planning gives a median area difference of 138. The difference between opponent models is somewhat lower than the median difference between the planning seeds. This lower difference is not surprising because the evaluation of a path depends strongly upon the starting positions of the agents. The seeds are designed to roughly cover all possible starting positions of the opponents, so the variation of the hillclimbing seeds should be higher.

Plan Set	Median Area Difference	# Comparisons
Planning seeds	315	138
Across opponent models	138	223
Within one opponent model	0	6625
Within one opponent model (unique plans)	116	3073

Table 6.5: Median area differences among several sets of plans. The area difference roughly captures the variation among the plans.

[5]We use median rather than mean because the distributions of path areas have heavy tails.

The variation of the plans observed by using two different models for planning is also higher than the variation observed for using just a single model. We ran the planner on a set of 25 problems 53 times in order to understand the variation in the plan returned by the hillclimbing approach. Not surprisingly, the median area is 0, since the same plan is returned many times. If we restrict our attention to just the set of unique plans[6] returned over these 53 trials, the median area of 116 is still smaller than the median area between plans returned by different models. This result suggests that, as expected, the model used causes more variation in the output of the planner than the random variation in the hillclimbing.

6.4.2 Total Game Effect

A team's overall performance in a simulated soccer game is a product of many tightly interacting factors. The experiments in this section are designed to show that adaptive setplays can have a positive impact on the overall team performance. It is not a thorough evaluation of *when* and *why* the adaptive setplays have an impact. In other words, these experiments are an existence proof that the representation and algorithms that we are proposing can have a positive effect on the performance of the team.

Of the course of a simulation game, setplays are executed for a fairly small percentage of the total time. Therefore, the absolute effect of setplays on the final score difference is expected to be small even if the setplays are significantly better than what was present before. Any significant effect of the overall score of a team by improving the setplays is an achievement.

The plan execution algorithm was fully implemented in the ATT-CMUnited2000 simulation soccer team [Riley et al., 2001]. The team ATT-CMUnited2000 was based on CMUnited99 [Stone et al., 2000] which had fixed setplay plans for each type of setplay (goal kick, kick in, free kick, etc.). We ran ATT-CMUnited2000 using the old fixed setplays and the new adaptive setplays, playing against CMUnited99 in both cases. The results are shown in Table 6.6.

The results show that the new setplays have a small but significant effect on the overall performance of the team. The effects on goals against seems to occur for two reasons. First, setplays such as goal kicks and goalie catches can be dangerous times for the team because the ball starts so close to the goal. Executing a good play to get the ball upfield can get out of these dangerous situations. Second, a good offense can be the best defense. If the team spends more time attacking, the opponent has less opportunities to score goals. This result is good given the small proportion of the time of the game occupied by setplays.

[6] At most 2 different plans were returned for a given model and problem.

	# Games	Mean goals scored	Mean goals against
CMUnited99 fixed setplays	33	2.45	0.30
Planned, MASTN based setplays	56	2.55	0.18

Table 6.6: Comparison of fixed setplays from CMUnited99 to adaptive setplays using MASTN. With a standard one tailed t-test, the difference in goals scored is not significant, while the difference in goals against is $(p < .01)$. All games were against CMUnited99.

We wanted to further test the effectiveness of the adaptive setplays. However, there is considerable effort in linking the plan execution algorithm to the behavior architecture of a player. The standard coaching language CLang (see Section 2.2.2) was created around the time we were finishing the previous experiments. We therefore created an algorithm to convert the MASTN into CLang condition-action rules. This conversion allows the setplays to be used with any team that understands CLang rather than being restricted to ATT-CMUnited2000.

Translating the MASTN plan into CLang requires that the effects of actions are encoded into conditions of rules. Since CLang supports conditioning on ball and player position, this is in general possible. However, some aspects of the execution algorithm can not be encoded because of limitations in CLang. For example, since CLang has no action representing where an agent should be looking, the agent can not be told to look to where a relevant action should be taking place.

We use the coachable team ChaMeleons from Carnegie Mellon [Carpenter et al., 2002] as the recipient of our planning advice. The opponent used is Gemini from the Tokyo Institute of Technology [Coradeschi and Tadokoro, 2002]. We used Gemini for two reasons. First, Gemini was the opponent in the RoboCup 2001 coach competition, allowing us to compare performance to what was observed there. Second, since Gemini was not created by us, nor do we have any knowledge of its behavior algorithms, this provides a more independent test of the setplays.

In order to assign roles in the plan, the coach needs to know the formation, or arrangement of players on the field, in order to assign closest players to the roles. Therefore, our coach also sends the players a formation, i.e. a spatial assignment. Details about the structure of the formation and how it is learned can be found in Section 5.2.1.

We ran a series of simulation games under different conditions.[7] Each experimental

[7] In all of these experiments, we slowed the server down to 3-6 times normal speed so that all agents could run on one machine. This was done for convenience for running the experiments. We tried to verify that agents were not missing cycles and while this setup shouldn't affect outcomes compared to running on

condition was run for 30 games and the average score difference (as our score minus their score) is reported. Therefore a negative score difference represents losing the game and a positive score difference is winning. All significance values reported are for a two tailed t-test.

Three sets of games were run: a baseline without the coach, the coach just sending a formation, and the coach sending a formation and planning setplays. Since there are many interacting factors affecting the performance of a simulated robot soccer team, we are more interested in the improvement that the setplays has on the coached team rather than the absolute win/loss value of the coached team against the opponent. Table 6.7 shows the results.

Condition	Score Difference
Baseline (without setplays)	-6.5 [-7.2, -5.9]
With formation	-9.1 [-10.0, -8.2]
With setplays and formation	-4.2 [-4.9, -3.5]

Table 6.7: Mean score difference under various experimental conditions. The score difference reported is coached team score minus opponent score. The interval next to the score is the 95% confidence interval.

The use of the setplays significantly ($p < 0.01$) improves the performance of the team, both over just the use of the formation and over not using either a formation or setplays. The effect here is larger than in the previous experiment which compared fixed vs. adaptive setplays in ATT-CMUnited2000 since we are comparing no set plays to our adaptive setplays in this case.

6.5 Summary

This chapter has described Multi-Agent Simple Temporal Networks, a plan representation that, when combined with the execution algorithm presented, provides for distributed execution of team plans. We have also presented a plan generation algorithm for the simulated robot soccer environment.

The generation and execution algorithms are fully implemented in a simulated soccer coach and team. Empirical results show the positive effects of the setplays on the overall performance of the team, as measured by goal difference. Additionally, we have created

several machines, the design of the server makes it impossible to say for sure.

an algorithm to translate MASTNs into CLang (Section 2.2.2) rules, which allows us to test the setplays with a team not originally designed to execute the plays. Once again, the empirical results show that the setplays can improve the overall team performance.

Chapter 7

Relation to Previous Work

The coaching problem touches on many other areas of research. This chapter discusses the relevant related work and how this thesis improves upon or addresses new problems or dimensions in each of these areas. Section 7.1 discusses related work on how agents can incorporate advice (typically from humans) into their decision making. Section 7.2 then discusses agents generating advice, including Intelligent Tutoring Systems (where humans receive advice) and formal models for classification learning with a teacher. Section 7.3 discusses agent modeling, one of the important techniques used in this thesis. Section 7.4 discusses related work in coaching in simulated robot soccer, the motivating environment for most of this thesis work. Section 7.5 discusses relevant work in learning and using Markov Decision Processes, which are the models used in Chapter 4. Section 7.6 discusses other algorithms that analyze behavior traces. Finally, Section 7.7 discusses multi-agent plan representations.

7.1 Agents Taking Advice

A variety of work has been done on using advice (often from humans) to improve autonomous agent performance. While this thesis has focused on advice generation, advice taking comes up in several places.

In Chapter 3, the predator agent in a predator-prey environment is receiving advice from a coach. The predator is a reinforcement learner, quite similar to Clouse [1995]. The primary difference is that our Q-learner does not receive reward simply for following the coach's recommended action. Another modification of an advice taking Q-learner is to discard state information from before advice is applied, with the assumption that the coach gives advice when recent actions were wrong [Clouse and Utgoff, 1992]. Lin [1992] presents another alternative where an entire sequence of actions is taken, along with the reward expected by the coach. Another alternative has been used in the RoboCup mid-size league. Takahashi et al. [2004] use human demonstrations as recommended actions to seed the action policy of a reinforcement learner.

In a different area of reinforcement learning, Driessens and Džeroski [2004] present a thorough study of incorporating guidance (in the form of example trajectories) into *relational* reinforcement learners. A relational learner is one where the value functions and policies are represented by logical formalisms such as first order logic instead of by tables or statistical function approximaters. Across several domains and with several different learning algorithms, the authors discover that guidance does help, but the parameters of when and how much guidance to provide varies with the underlying relational learner.

An alternative approach to incorporating advice into a reinforcement learning is provided by Wiewiora et al. [2003]. The reward function is modified using variants of potential based reward shaping [Ng et al., 1999]. They present two ways to incorporate advice, both of which have limitations. With "look-ahead" advice, the agent's learned Q-table must be post-processed to extract the optimal policy. With "look-back" advice, the learner must use an on-policy algorithm and the authors suggest that the learning time will be longer.

The algorithms in Chapter 3 are another alternative among the many approaches to incorporating advice into a reinforcement learner. All of these approaches are generally successful in using advice to improve reinforcement learning speed.

A closely related research area is that of imitation. Similar work has gone under many different names: learning by demonstration [Kaiser et al., 1995, Bakker and Kuniyoshi, 1996, Atkeson and Schaal, 1997], behavior cloning [Sammut et al., 1992, Šuc and Bratko, 1997], learning by watching [Kuniyoshi. et al., 1994], and behavior or agent imitation [Dautenhahn, 1995, Price and Boutilier, 1999, 2000]. In all the cases, the robot or agent is given example(s) of some task being done successfully (often from a human demonstration), and the agent's goal is to perform the same task. The demonstration of the task can be seen as a set of advised actions for the agent, quite similar to Lin [1992].

Imitation is used in Section 5.2. The coach uses example executions to construct team models. The content of these models is then provided as advice for the agents' execution. From this perspective, the coach modeling and advice provides yet another scheme for imitating, though unlike the approaches above, the imitation occurs for an entire team of agents.

7.2 Agents Giving Advice

In addition to agents receiving advice, past work has considered agents generating advice. One group of past research is Intelligent Tutoring Systems. These systems are designed to help human students learn, much like a private human instructor. The architecture of these systems is fairly standard at this point [Polson and Richardson, 1988]. The three primary components are:

Expert This component is a model for how the task should be performed. The expert could, for example, be a program that performs in an ideal manner or be a specific task structure that the student is supposed to learn.

Student Model The goal of this component is to understand and track what the student knows, based on the student's interaction with the system.

Tutor During learning, the expert and student model will have discrepancies. The tutor provides advice, questions, and other output to try to move the student towards the behavior of the expert.

The user interface between the student and the tutoring system is also an important component, but since there it has less direct resemblance to the coaching of automated agents, I will not examine this issue here.

One significant problem in tutoring systems has been the rigidity of the expert module. In cases where a specific, declarative task must be learned [e.g., Rickel and Johnson, 1999], an expert which simply has the "right answer" can be effective. This simple form of an expert module is quite different from our coach agents that must learn about the task in order to provide advice.

Creating a good student model that can accurately track a student as he learns has been a major endeavor for cognitive psychology [e.g., Anderson et al., 1990]. Knowledge tracing [Corbett and Anderson, 1995] deals with how to estimate what concepts a student knows based on simple correct and incorrect answers to quizzes. In this thesis, a similar problem is faced by the coach, which must estimate what the advice receivers can do (see Chapter 3). We use a simple scheme based on Q-learning. While this Q-learning scheme proves to be effective for the environment we consider, the number of repetitions required probably make the techniques presented here less applicable to humans.

Intelligent Tutoring Systems have been developed in too many application areas to list here, but most train for solitary problem solving. Recently, automated training systems for team tasks have gotten some attention [Miller et al., 2000, Rickel and Johnson, 2002].

The automated agents can function both as team members and as coaches for the trainees. Coaching for team tasks has an added dimension that the student must learn to think about what the other agents are doing. In order to guide the student to considering the correct things about the other agents' states, the coach needs a greater ability to monitor and advise about the student's internal state.

The idea of an agent teacher has also attracted attention from machine learning theorists [Goldman et al., 1993, Goldman and Kearns, 1995, Shinohara and Miyano, 1991, Jackson and Tomkins, 1992, Goldman and Mathias, 1996, Mathias, 1997]. The framework considered is usually some variant of a teacher presenting examples of a target concept to a learner who is trying to classify them. The primary problems addressed are definitional. How does one define a good teacher? What if a particular teacher teacher only works with one particular student? These questions have not, to this point, been especially important for us to consider as we measure coach performance by testing with a variety of agents. As we move to more rigorous and first principles based evaluations, these questions may become more relevant. However, the theory framework does not currently match our coaching problem very closely, primarily because we are not working with classification tasks and we use a richer action based advice language.

7.3 Agent Modeling

Agent modeling has been an important topic in computer science and the literature available is correspondingly vast. Since one of the aims of this thesis is to examine how opponent modeling can be applied to the coaching problem, this section will briefly discuss some of the work in agent modeling, especially as it pertains to adversary modeling and adaptation. We'll briefly review approaches for agent modeling in simple turn taking games and repeated matrix games before drawing distinctions with our work.

In turn taking, perfect information games such as checkers and chess, pruned brute force search of the minimax tree combined with intelligent, domain specific evaluation functions has been the standard technique, such as in checkers with Chinook [Schaeffer et al., 1992]. However, in cases where the opponent is not the optimal player, better performance may be achieved through modeling the opponent. Carmel and Markovitch [1996] present M^*, a modified minimax search incorporating a model of the opponent. Further, Billings et al. [1998] argue that in order to play games that involve bluffing and partial information such as poker, opponent modeling will be necessary to achieve high performance. Their system uses a parameterized version of the agent's internal evaluation function to estimate the way the opponents value their hands. That information is then used in the search and evaluation of moves.

Repeated matrix games have also been a popular domain for considering opponent modeling. Carmel and Markovitch [1998] present the US-L* algorithm to infer a finite automaton that explains the opponent's behavior, and then find an optimal response. Gmytrasiewicz and Durfee [1995] formalize recursive modeling where each agent is considering what the other agent thinks. This framework has also led to interesting complexity results for trying to play nearly optimally against some fixed set of strategies, such as Turing machines, boolean functions, or simple functions of simple history statistics [Fortnow and Whang, 1994, Freund et al., 1995]. The notion of "nearly optimal" requires careful definition and different variants are used for different purposes. There are some positive results, such as Freund and Schapire [1999] who present an algorithm whose reward received is close to what could have been received from the best fixed strategy.

From a high level, these models answer the coaching sub-question, "How can models be used to determine desired actions for agents?" The models in Chapter 5 are used to answer the same question. However, the coaching environments we consider in this thesis generally have continuous state and action spaces as opposed to the discrete spaces discussed above. Therefore, the model representations must be very different and we have a more relaxed definition of success.

Modeling has also been applied in larger, real-time domains with continuous state and action spaces. Using an agent's own behavior representation to infer opponent actions has been used by Tambe and Rosenbloom [1995] in an air combat domain and by Laird [2001] in Quake, a real-time first person shooter computer game. We explicitly avoid having our agent models be based on the agent's internal execution policy because the coach is not an agent which executes actions in the environment and we want to provide as much generality as possible in our models.

Work on agent modeling has also occurred in the simulated robotic soccer community. Wünstel et al. [2001] use self organizing maps to classify the movements of agents. Miene et al. [2004] use coarsely discretized state and action descriptions to arrive at a "qualitative" motion model which can be used to predict impending offside situations. Also in RoboCup, but for the small size robots, Han and Veloso [1999] use Hidden Markov Models (HMM) to recognize behaviors of robots, with each HMM representing a model of a behavior of a robot. These models are focused primarily on recognizing the behavior of an agent or team and not on how to use such a model to improve performance.

The ISAAC system [Raines et al., 2000] analyzes past games to look for patterns at both the individual agent level and the team level (see Section 7.4 for more on ISAAC). These models have similar structure to the passing models presented in Section 5.2.2.

Kaminka et al. [2003] also create team models to be used in predicting the future actions of a team. Events like passing and dribbling are extracted from the observation data and are then considered as symbol sequences. Repeated patterns in these sequences are found and used for prediction. The primary difference with our work is representational; our models are based on clusters and decision trees and their work uses compact representations of repeated subsequences.

Section 5.1 argues that selecting between opponent models online rather than trying to learn from scratch is a promising avenue for dealing with sparsity of data in online setting. This same idea is discussed and tested in a predator-prey domain by Denzinger and Hamdan [2004]. Their models, which they call "stereotypes," have a very different structure and use different selection algorithms, but end goal is the same.

7.4 Coaching in Robot Soccer

Coaching fits naturally into the format of robot soccer. To my knowledge, one of the first teams to use an online coach was Kasugabito [Takahashi, 2000]. Based on the score difference, time remaining, and ball's path, the online coach would adjust the team's formation.

Since then, formation learning and switching has been a popular approach. Our hill-climbing based learning of formations is presented in Section 5.2.1. The Sharif Arvand team uses standard image processing algorithms on windowed histograms of player location to find regions where players frequently are [Habibi et al., 2002]. Virtual Werder [Drücker et al., 2001] uses a learned neural network to recognize formations and then a hand-coded decision tree to change the team's own formation. Our work predates all of these approaches. While there are differences among these algorithms, all have the same goals and perform reasonably well.

Besides the opponent modeling work described in Section 7.3, the other important predecessors to online coaching systems were the online commentator systems. ROCCO, BYRNE, and MIKE [Andre et al., 2000, Tanaka-Ishii et al., 1998] are three of the first systems. While there are important differences in system structure, presentation, and dialogue management, all the systems track various game statistics to match observed behavior to pieces of dialogue. This understanding of the game from observation is an important component of many coach systems.

One of the first systems to analyze a team's play to provide advice was ISAAC [Raines et al., 2000]. ISAAC used decision tree learning to identify rules describing the conditions for when goals were and were not scored for and against a team. ISAAC had no automated way for the agents to incorporate advice; the output was descriptive natural language which human developers could use to change their team. This thesis improves on this work by providing easily operationalizable advice for similar style models.

Online coaching with automatically incorporated advice gained importance in RoboCup 2001 with the creation of the online coach competition. The community (including myself) created the language CLang (as described in Section 2.2.2) to allow coaches and teams to inter-operate. The language has been used in all subsequent coach competitions. UNILANG [Reis and Lau, 2002], a competing attempt to create a standard language, did generate some ideas which were drawn into CLang. In addition to the research presented throughout this thesis, the other team from that time which published opponent modeling research results was the Dirty Dozen [Steffens, 2002]. The focus of the work was Feature-Based Declarative Opponent-Modeling (FBDOM) where opponent models use features which are associated with actions. In contrast to most of this thesis, Steffens's models are created by hand. These models can then be matched to observed behavior similar (as in Section 5.1.2) or used to imitate a team (as in Section 5.2).

In addition to the formation learning and adaptation techniques mentioned above, I know of two other coaches which make use of machine learning. The UTAustinVilla coach [Kuhlmann et al., 2004] learns and uses similar opponent models to ones described in Section 5.2, though with some important representational differences. Namely, while they do use decision tree learning, they do not perform the clustering step that we do. Also, they use a much larger set of features including distance to the goal and the current score. The Sharif Arvand coach [Ahmadi et al., 2003] uses a two-layered case based reasoning approach to predict the future movements of the player and the ball, though it is not specified in that paper how predictions are translated into useful advice.

7.5 Markov Decision Processes

Markov Decision Processes (MDPs) [Puterman, 1994] have been one of the most used and most studied formalisms in artificial intelligence research. Chapter 4 continues in this line by presenting a set of algorithms to learn an abstract MDP from observed executions.

In this process, a Markov Chain is first constructed and then an MDP is formed from the Markov Chain. Shani et al. [2002] also explain a mechanism for Markov Chain to Markov Decision Process conversion. However, for them the Markov Chain describes how the system acts without agent actions. They use a simple proportional probability transform to add the effects of the actions. In this thesis, the Markov Chain represents the environment with agent actions and we need to infer what those actions are.

In Chapter 4, the state abstraction function uses a factored state representation. The AND/OR tree structure is one simple example of using a factored representation to make reasoning easier or more efficient. Dean and Kanazawa [1989] give the basic representation of a factored MDP as a Bayes net. A number of authors then give algorithms to use this sort of compact representation to do more efficient reasoning [e.g., Boutilier and Puterman, 1995, Koller and Parr, 1999, Guestrin et al., 2001, Boutilier et al., 2000]. These techniques have also been extended to multi-agent games [Koller and Milch, 2001] and distributed planning [Guestrin and Gordon, 2002]. Our transition classification scheme described in Section 4.3.3 is another use of a factored state representation.

The AND/OR tree can also be viewed as one structure for representing context-specific independence (CSI). In terms of Bayes nets, CSI means that two variables are independent given *some* values of another variable. Note that in a standard Bayes net, the lack of an arc means that the two variables are (conditionally) independent for *all* values of the other variables. Geiger and Heckerman [1996] present Bayesian multi-nets as a way to represent CSI. A series of separate Bayes nets are constructed and the difference in the edges of each network represents the independencies. Boutilier et al. [1996] build on this work to present ways that such dependencies can be discovered and utilized for normal Bayes nets where the conditional probabilities at the nodes are represented by trees. Chickering et al. [1997] then presents a heuristic search for finding such representations given data. The AND/OR trees are an easier to use but less expressive manner of representing CSI. An OR represents that if one operand factor has a non-null value, it is independent from the other operand factor.

In this thesis, the state abstraction is given to the system *a priori*. The learning of a good state abstraction has long been a pursuit of AI and we will not attempt to provide a survey here. However, use of any of these techniques in order to learn the abstract state space while learning the transition model (as suggested in Section 8.4.1) is an interesting direction for future work.

The abstract MDPs learned in this thesis also use temporally extended abstract actions. Abstract actions have a long history, especially in the planning literature [e.g., Korf, 1985]. Recently, several formalisms for abstract actions have been presented for MDPs, notably in the context of reinforcement learning. All are based on semi-Markov Decision Processes (SMDPs) [Puterman, 1994]. SMDPs extend MDPs with a probabilistic duration in each state. Options [Sutton et al., 1999] are intended to augment an MDP with temporally extended actions. Each option includes a partial policy and probabilistic termination conditions. An MDP with options can then be modeled as an SMDP. By leaving the base level MDP actions in the SMDP, an optimal policy for the SMDP is guaranteed to be optimal for the MDP. Hierarchies of Abstract Machines [Parr and Russell, 1998] and the MAXQ

Value Function Decomposition [Dietterich, 2000] instead focus on incorporating domain knowledge in the form of subtask decompositions. These subtasks form a graph structure where each node represents a task and the policy for each task can only select between that task's children. By removing the base level actions, the state and action spaces at each subtask can be reduced, which makes learning much easier. However, the decomposition restricts the space of possible policies, and therefore a poor decomposition can lead to a sub-optimal policy.

All of these schemes explicitly model time taken for the abstract actions. This time is currently ignored in the MDP learning in Chapter 4. Extending the learning process to use any of the above formalisms is a good direction for future work (see Section 8.4.1).

7.6 Learning from Execution Traces

An important part of a coach agent as described in this thesis is the ability to analyze a trace of past behavior and learn from it. Phrased in this general way, many other learning algorithms solve a similar problem. In this section we will discuss two such areas that learn what could be considered advice: control rule learning and chunking. We will first briefly review these areas and then discuss how coaching differs from both.

Many AI systems perform some sort of search, and in these systems, domain-specific search control heuristics, sometimes called control rules, can help improve the speed of search or the quality of the solution returned. Various approaches have been given to learn such control rules. In the LEX system [Mitchell, 1983], which solves integral equations in closed form, Explanation-Based Generalization [Mitchell et al., 1986] is used to process the trace of a problem solving episode, along with a complete and correct domain theory, to produce search control rules that are guaranteed to be correct. Minton [1988] takes a similar approach in the PRODIGY [Veloso et al., 1995] planning system. Other approaches [e.g., Borrajo and Veloso, 1996, Estlin and Mooney, 1997] relax the requirement for complete and correct domain theories and learn possibly incorrect control rules which are then refined and corrected as needed.

Chunking originally came from psychology as a model of learning in human and animals (see Laird et al. [1986a] for background). The basic idea is that as an agent goes through experiences, repeated patterns of symbols can be "chunked" into one symbol, allowing better or faster recall or execution.

The SOAR architecture [Laird et al., 1987] is intended as a model of general cognition. All reasoning takes place as search in problem spaces. This uniformity of execution allows a single learning algorithm, namely chunking, to learn in all parts of the system [Laird et al., 1986b, 1984]. At the completion of a problem solving episode, the goal and solution are considered as a candidate chunk, with much of the same generalization reasoning as goes into control rule learning.

In both control rule learning and chunking, the reasoning is very closely tied to the decision making process of the system they are trying to improve, even as far as having a complete and correct logical model of the system [Minton, 1988]. Coaching provides an abstraction of the decision making process through the concept of advice. Because of this abstraction, different agents can successfully implement and use the advice, though more research remains to be done on understanding how much of the underlying agents needs to be understood by the coach (see Section 8.4.2). Further, because of this abstraction layer, the coach does not have access to the full reasoning process of the agent.

7.7 Multi-Agent Planning

Chapter 6 introduces Multi-Agent Simple Temporal Networks (MASTN) as a plan representation for execution by distributed agents. MASTNs refine Simple Temporal Networks (STNs) to be used in the multi-agent case. STNs have been used to solve scheduling problems [e.g., Morris and Muscettola, 2000] and the execution algorithm we present uses the algorithms presented by Muscettola et al. [1998] as subroutines.

Other researchers have suggested representations for multi-agent plans. A number of models of multi-agent systems have been proposed recently [Boutilier, 1999, Peshkin et al., 2000, Bernstein et al., 2000, Xuan et al., 2001, Pynadath and Tambe, 2002a]. For all of these models, solving the model yields a joint action policy for the agents. This policy can be seen as a universal plan for the agents. However, the dimensionality of the model means that such polices quickly become difficult to construct or communicate. Therefore, a naive use of the models would not be an effective way to create and communicate about team plans. Indeed, one of the motivations for creating the COM-MTDP model [Pynadath and Tambe, 2002a] was to be able to evaluate algorithms which use more efficient reasoning processes. The MASTNs that we introduce are one compact way to represent a joint plan.

Bowling et al. [2004] introduce tactics, plays, and play books as multi-agent plans. In the context of small size robot soccer, they define tactics as single agent, primarily reactive behaviors. Plays are specifications of roles by the sequence of tactics they should be performing. The play book is a collection of plays. This approach addresses a number of orthogonal issues to those addressed by the MASTNs. Most notably, the strength of

MASTNs is in handling agents whose beliefs about the world may be inconsistent and allowing them to coordinate successfully. While the play-based approach must consider multi-agent problems such as role assignment, one central controller makes those decisions.

Teamwork theories must also consider representing and manipulating multi-agent plans. SharedPlans [Grosz and Kraus, 1996] and Joint Intentions (as expressed in GRATE* [Jennings, 1995]) are normative specifications of mental attitudes of agents in a team. These teamwork theories are meant to be used to guide the design of agents. Both use the concept of a recipe where agents are committed to performing possibly ordered actions. While both approaches catalog recipe failures and specify how to deal with them, MASTNs provide a more specific framework for representing and reasoning about coordination and failure through temporal constraints. In particular, those approaches specify how the beliefs, desires, and intentions should change in response to team events, but the approaches leave open how those beliefs, desires, and intentions are translated into actions. Therefore, while the SharedPlans and Joint Intentions frameworks are more general, MASTNs provide more of a solution if one can represent the needed coordination and failure modes in the temporal constraints of the network.

STEAM [Tambe, 1997] draws on both SharedPlans and Joint Intentions. The key innovation is the introduction of team operators. Each agent maintains its own perception of the state of execution of these team operators. STEAM is then a system for maintaining as much consistency as needed and possible among these operators. Thus, our MASTN representation and algorithm can be seen as an application of these same concepts into a representation with temporal constraints.

Intille and Bobick [1999] also use temporal constraints to express coordination. However, their temporal constraints are fuzzy and qualitative, such as “A around B” meaning that event A should occur around the same time as B. The other major difference is that the nodes in their temporal network represent agent goals, not particular events. They apply their representation to plan recognition in records of human American football. In other words, similar temporal constraints and reasoning are used to solve a different problem.

Chapter 8

Conclusion

Coaching is a well known relationship among humans. This thesis has explored a range of issues that arise when one automated agent is coaching another. Coaching between automated agents is a recent development and this thesis helps to define and develop this largely unexplored area. The breadth of the coaching problem is reflected in this thesis. Further, this thesis represents many years of experience in working with agent coaching agent systems.

This chapter reviews the scientific contributions of the thesis (Section 8.1), discusses some insights into coaching through my long experience in this problem domain (Section 8.2), considers the insertion of knowledge into the coach agents (Section 8.3), and presents some of the promising future research directions that build on this thesis (Section 8.4).

8.1 Contributions

This thesis makes several important scientific contributions:

- **A study of advice adaptation under various limitations, empirically tested in a predator-prey environment.**

  One important task of a coach is to understand the limitations of the agent(s) receiving advice. A first step towards addressing this problem was made through experimentation in a predator-prey environment. Several simple learning algorithms based on Q-learning were given for the advice-receiving predator agent. All of these algorithms incorporate advice in the form of a recommended action from the coach. Coaching algorithms of varying complexity that generate this advice were shown. Limitations on the predator's actions, the communication bandwidth, and the predator's memory were explored and it was shown that the coach could learn about the agent's limitations. Limiting communication and memory predictably reduced the predator's performance. The experiments support the idea that the more limitations, the more important it is for the coach to learn in order to provide more effective advice.

- **A set of algorithms for learning an abstract Markov Decision Process from external observations, a given state abstraction, and partial abstract action templates.**

  A coach must have knowledge in order to provide useful advice. We contribute one set of algorithms for a coach to acquire useful knowledge of the environment. The coach observes past executions of agents in an environment. Notably, those executions contain no explicit information about the actions taken by the agents. From those executions and a given state abstraction, the coach constructs a Markov Chain. The state abstraction is factored, and the factors are combined with an AND/OR tree that we introduce. The Markov Chain represents how the environment changes while agents are acting. From the Markov Chain and given abstract action templates, a Markov Decision Process is constructed. These abstract action templates contain no probabilities. They do specify sets of primary and secondary transitions. Primary transitions represent the normal or intended effects of an action and secondary transitions represent other possible results.

  Given the transition model in an MDP, we can add arbitrary rewards, solve the MDP, and generate advice based on the optimal policies. The algorithm is not specific to the robot soccer domain. We show that this learning process is effective at producing useful advice in several variants of simulated robot soccer and in the RCSSMaze environment which we constructed.

- **Several opponent model representations, learning algorithms for those models, and advice generation algorithms using those models.**

  Three different models of team behavior were created. The first probabilistically predicts opponent movement in response to the planned actions of our team. While these models were written by hand, an algorithm was given to select online between a set of predefined models. The effectiveness of this algorithm was shown experimentally.

The second model represents the general spatial relationships of a team of agents through a "formation" in simulated robot soccer. The formation of a team is learned through observations of past performance. When playing a given opponent, the formation of a previously successful team can be imitated. The learned formations were shown to make a significant difference in a team's performance.

The last of the models captures patterns in the actions of a team. Using clustering and decision tree learning, models which capture the passing patterns of a simulated robot soccer team can be learned from observations of the team playing. Advice can then be generated to predict an opponent's behavior or to imitate a team that was successful. While the advice thus generated was not shown to have a large impact, the models do appear to be capturing behavior patterns of a team.

- **Multi-Agent Simple Temporal Networks (MASTN): a novel multi-agent plan representation and accompanying execution algorithm.**

  Another important question in a coaching relationship is how the advice is structured. Throughout much of this thesis, we used the CLang advice language created by the simulated robot soccer community (including myself). However, making a language that is easy to use and effective for the agents is an important topic. We contribute MASTNs as one particular language for advice as plans for distributed execution. MASTNs refine Simple Temporal Networks, using temporal constraints to express multi-agent coordination. We also contribute a distributed execution algorithm which allows the distributed agents to easily handle plan coordination and detect plan failure.

  A plan generation and execution system is fully implemented in the simulated robot soccer domain. We experimentally show that the use of the plans has a positive effect on the performance of several teams.

- **Largest empirical study of coaching in simulated robot soccer.**

  This thesis is heavily experimental and motivated by the idea of coaching in simulated robot soccer. Thorough experimentation has been carried out in a variety of experimental conditions. The experiments in this thesis represent over 5000 games of simulated robot soccer and were generated with over 2500 hours of computer time. In contrast, the total number of official games from all RoboCup competitions between 2000 and 2004 is only 1223. This data is all available upon request (see Appendix C for details) and provides a valuable resource for researchers in this area.

8.2 Qualitative Lessons

In addition to the possible future research directions discussed in Section 8.4, there are some qualitative lessons about coaching environments that may be useful to others. While these are not fully substantiated scientific claims, I believe that these insights can help others to further understand the coaching problem.

- **The coach and advice receivers are a tightly coupled system.**

  While part of the appeal of having a coach agent is to modularize the decision making process of agents in the environment, the coach must be able to fundamentally affect the behavior of the agent. Often in computer systems, and perhaps more so in agents, there are dependencies of which the designer is not aware. Incorporating advice can reveal these unarticulated assumptions and dependencies. Simply specifying a language and then separately developing a coach and advice receiver is likely to lead to a poorly performing system when they are first brought together.

- **Coach learning will require iteration to achieve the best performance.**

  Much of the machine learning that takes place is a sort of "one-shot" learning. Some training data is supplied, some intelligent algorithms are applied, and then the result is evaluated. Coaching will require iteration over the learned models, either with a human in the loop or without, to achieve the best performance. For example, in the RCSSMaze environment, I had originally made an error in the transition specifications for an action. This error resulted in the agent moving directly into a wall on almost every trial. Both human provided and learned knowledge must be subject to revision and improvement through iteration. Coaching should be a continual interaction and not one-shot advice. Iteration was naturally done during development and in the RCSSMaze environment we explored somewhat the results of learning an MDP from a previous MDP's execution.

- **A tradeoff exists in how much of the state space to cover with advice versus how good the advice is.**

  The choice is faced by both the human designer and the coach agent. For example, in the MASTN work, the coach is only giving advice for the few seconds of our setplays and gives no help for the rest of the game. However, the representation is specialized for this task and is quite effective. The MDP based advice attempts to advise about a much larger part of the game. However, restrictions in the amount of data, amount of communication bandwidth, and language structure limit how specific the advice thus generated can be. Less specific advice may help the agents less by either providing less guidance or grouping together states which really should be receiving different advice. While simultaneous improvements on both dimensions are certainly possible, finding the correct balance is an important part of creating an effective coach agent.

- **Different observability by the coach and agents can be ignored somewhat, but will need to be considered at times.**

  In much of the work here (with the MASTNs for setplays being the exception), the coach was virtually ignoring the fact that the agents had less than perfect knowledge of the world state. Partially, this was successful because, as a domain expert, I chose for the coach to use world features that the agent would probably know. However, the partial observability can cause problems. In an earlier version of one of the RCSSMaze 0 experiments, the agent continually had problems performing an unadvised action at a particular point in the maze. It turned out that the agent did not know one of the walls was next to it and therefore matched the world state to the advice incorrectly. It is not clear at this point how much of an effect partial observability had on the other experiments. The coach's learning systems were greatly simplified by ignoring the problems of partial observability and effective advice was still generated. However, it is not clear if this will continue to work.

- **Analyzing the past behavior of an agent is most useful only if the future will look similar to the past.**

  Advice is useful to an agent only if it helps that agent in situations encountered in the future. However, the coach will generally analyze past performance data in order to produce advice. The implicit assumption here is that the future will look like the past. This may not always be true, especially in the short term. In a soccer game, if, after a goal is scored, a coach advises the players about how they should have played when the score was tied, the advice may not help short-term performance. Similarly, in the predator-prey environment we encountered the problem that the advice about one particular state just experienced was just as likely to be used by the agent as advice about any random state.

8.3 Insertion of External Knowledge

At various points throughout the thesis, a domain expert (namely, the author) provided knowledge to the various coach agents. Credit for the success of the coaches must therefore be given not just to the algorithms presented, but also to the quality of the knowledge provided. Thorough studies of the exact effect of providing better or worse knowledge to the system are outside the scope of this thesis, but it is important to acknowledge that external knowledge was inserted. This property is in no way unique or even unusual among AI systems. Based on previous experience, the designers of AI systems must make many choices.

This section briefly discusses four types of knowledge that were given to the system. We will not cover every instance, but this discussion should help the reader better understand what it takes to successfully apply the algorithms given in the thesis.

Representational Choices The choice of representation almost always restricts the space of models that can be considered. For example, we see this with the choice of Gaussians for the probability distributions in Section 5.1.3, the use and conversion of Autoclass C clusters in Section 5.2.2, and the selection of features for the rule learning in Section 5.2.2. These choices were made because we believed that the models which could be expressed in these representations were sufficient for the tasks.

Abstractions In several places, we provide the coach with abstractions to use in reasoning about a problem. The abstract state and action spaces discussed in Chapter 4 are the most obvious examples, but the choice of events for the planning in Chapter 6 and the particular models in Section 5.1.4 are also abstractions that were provided to the coach. While some general formats and representational tools are given, providing good abstractions still requires knowledge of the domain.

Training Data A very important, but sometimes ignored, aspect of a learning system is the choice and quality of the data provided to the system. This aspect is especially important in the team models in Section 5.2, where we chose a particular team to imitate. As demonstrated with the environment models in Section 4.4.3, the properties of the training data can greatly affect the quality of the learned model. In some cases, one simply has to use whatever data is available, but the importance of the training data quality on overall effectiveness can not be ignored.

Algorithm Parameters One of the most obvious places where external knowledge is applied to the system is in the choice of parameters to the algorithms. The machine learning literature is full of algorithms with numerous parameters that must be set "appropriately" for the problem at hand. At several points in this thesis, informal experimentation was done with algorithms to tune parameters.

Besides clearly acknowledging the use of external knowledge in the coach agents, this discussion allows us to make two conjectures about knowledge insertion. First, even though some domain specific work may have to be done (by providing knowledge in the forms above), we believe that the framework and algorithms of this thesis allow an agent designer to more efficiently create an effective multi-agent system than if they had to start from scratch. Second, the boundaries of the system can be further expanded in the future. Each instance of knowledge insertion above is a candidate for future learning. With appropriate future work, a coach may be able to learn the required knowledge rather than having it provided. This possibility is explored in several places in the next section.

8.4 Future Directions

Agent to agent coaching is still a relatively new area in AI and this thesis opens a variety of future work directions. We present some future research directions expressed as possible thesis titles.

8.4.1 Abstract MDP Learning

The learning of an abstract MDP as a model of the environment (Chapter 4) proved to be useful to provide advice. However, there are a number of improvements which could be made.

- **Recursive Learning and Verification of Abstract Markov Decision Processes**

  Small errors in the MDP can cause a large performance degradation. Developing the correct abstract action templates required several iterations and minor tweaking of the system was needed to obtain the best performance. The potential exists to find and fix many errors in the learned MDP if the coach can experiment with and observe the MDP based advice being used. Recursively refining the MDP, possibly through learning processes similar to those already proposed, has great potential to improve the quality of the MDP model. While we experimented with this idea in the RCSSMaze environment in Section 4.4.3, the design and evaluation of better learning algorithms is a very interesting future direction.

- **Learning Hierarchical Semi-Markov Decision Processes from External Observation**

  The environment models learned in this thesis are abstract MDPs. However, as discussed in Section 7.5, even if the lower level can be modeled as an MDP, using abstract actions generally requires a *Semi*-Markov Decision Process (SMDP).

Unlike an MDP, an SMDP includes a notion of time taken by the various actions. Notably, these durations affect the time-based discounting. In the simulated soccer environment for example, the learned MDP goes through many more transitions while dribbling as compared to passing even though they may take approximately the same amount of time in the game. The extension of the learning algorithms in Chapter 4 to handle time in a SMDP, possibly using one of the formalisms described in Section 7.5, merits further attention.

In addition to handling the time taken by actions, handling hierarchical actions would be useful. Several of the abstract action formalisms are designed specifically to handle a hierarchy of abstract actions. Learning such a hierarchy and/or its parameters from external observation data is a potentially very useful extension of the work presented here.

- **Refining State Abstractions for Markov Decision Process Learning**

  In the MDP learning presented here, the state abstraction was given explicitly by the user. Past work in automatically learning state abstractions (see Section 7.5) could be useful in helping the user create such an abstraction. Further, some of those algorithms create an abstraction while learning to solve an MDP. Similarly, it may be possible for the state abstraction and transition model learning to take place simultaneously. If the transition learning could effectively identify places where the abstraction is too general or too specific, learning speed and accuracy could be improved. The abstract state learner could suggest refinements and abstractions to be considered by the transition learner.

8.4.2 Adapting to Advice Receivers

One challenging problem faced by a coach is the adaptation to the limits and peculiarities of a particular advice receiver. Chapter 3 began an exploration of how a coach could learn about agents' limitations and how the parameters of the environment affect the coach's learning, but much more work remains to be done.

- **Learning About Agents While Giving Advice**

  The coach in Chapter 3 had a simple Q-table based model of the actions that an agent could perform. Moving to more abstract advice and more complicated limitations will require better representations for reasoning about the limitations of an agent. For example, in the soccer environment, it is not that one team can't dribble while another can, but rather that the speed, safety, and overall effectiveness of dribbling will vary between teams. Creating domain-independent representations for agents' actions and their response to advice needs further research attention.

Also, in any reasonably complex advice language, there will almost certainly be differences in interpretation or just plain unsupported features among different advice receivers. Anyone familiar with the history of the coach competition at RoboCup knows that the various coachable agents have all had quirks in their interpretation of CLang. Differences in interpretation and other behavior oddities will likely occur in any reasonably complex advice receiving agent. Since one of the goals of coaching is to be able to work with different teams of agents, the coach will need some (preferably automated) way of learning about and adapting its advice to these differences in interpretation.

- **Talking Back: How Advice Receivers Can Help Their Coaches**

  Throughout this thesis, the communication has been one-way, from the coach to the advice receivers. The coach is assumed to get all of its knowledge about the agents by observing their behavior. However, any person who has been in a teaching or coaching role knows that it can be much more effective if the advice receiver understands how to ask for help or can articulate their confusions. Of course, a coach can not trust an advice-receiver to fully understand its confusions or to ask the right questions. However, it would seem that an agent could similarly be helpful to a coach. For example, the Instructo-Soar agent [Huffman and Laird, 1995] is able to ask a human for help or explanation. How could such algorithms be adapted to agent coaching agent scenarios? What should the agent to coach language be? How should the coach use the information from the agent? When should an agent ask for help or express confusion?

- **What I See and What I Don't: What a Coach Needs to Know About Partial Observability**

  As discussed in Section 8.2, partial observability for the agents has largely been ignored by our coaching algorithms. While this has been effective, understanding why it has and when it will continue to be are significant problems. Reasoning about the partial observability of the agent will almost certainly be more challenging for the coach. Under what circumstance is it worthwhile to do this computation? Should it be left to the advice receivers?

  One interesting point in the MASTNs of Chapter 6 is that the plan representation can tell the agent what part of the world that it is important to look at. Providing instruction on what to look at and what to notice is one interesting task that human coaches and teachers perform. How does this task translate to agent coaching scenarios?

8.4.3 Advice Languages

The advice language for the coach is one of the most important design decisions in a coaching system. This choice constrains what the output of the coach can be and has a large effect on the ability of the coach to produce useful advice.

- **A Taxonomy of Advice Languages**

  Throughout most of this thesis, we used the advice language CLang. While this advice language has served us well, an understanding of the space of possible advice languages is important. How language features affect such properties as expressibility, ease of implementation, ease of use/generation, and comprehensibility (by both humans and agents) are all important questions. Creating a formal model for expressing various properties of an advice language would aid in this task, but how such a model would be constructed is not clear at this point. Further, this thesis has focused on advice as recommended actions (or macro-actions). Are there other meaningful concepts for advice? How would these be encoded in an advice language?

- **Distributed, Multi-Agent Plans with Options: Negotiation and Coordination**

  MASTNs as presented in Chapter 6 are an interesting plan representation for distributed execution. However, while the plans express parallel actions, they do not adequately express options in the plans. That is, if centralized planning is followed by distributed execution, it would be useful to allow the plans to contain choices between qualitatively different paths such that those choices are made at execution time and not at planning time. However, adding such choices increases the burden on the executing agents to successfully negotiate and coordinate to choose between the options. Both representational and algorithmic challenges exist in creating and using such plans.

- **Seeing the Right Things in Multi-Agent Plan Execution**

  Similarly, the MASTN representation in Chapter 6 implicitly encodes some information about the perceptual expectations of the agents. That is, the agents expect to see the results of the various actions being executed. The plans are tolerant of the agents having differing beliefs, because of their partial observability, about when an action was executed. In some domains there may be actions or events which are not directly or immediately observable by the agents. If the plan representation explicitly encoded information [e.g., Doyle et al., 1986] about what the agents should and should not expect to see, then such actions could be handled, whereas MASTNs may have trouble. These encodings could potentially be nodes in the graph, allowing the agents to do the same sort of temporal reasoning as in STNs. However, exactly specifying how such perceptual expectations could be encoded and reasoned about is a significant challenge.

8.4.4 Opponent Models

The modeling of and adaptation to opponents is one way that a coach can provide useful advice to a team. In Chapter 5, we suggested several opponent models for simulated robot soccer. While these models do capture information about opponents in these complex environments, many improvements could be made.

- **Agent Models for Continuous State and Action Spaces**

  Many agent modeling techniques are designed for environments with discrete state and action spaces (see Section 7.3). The models we present in Chapter 5 are spatial models over continuous environments. While concepts like formations are applicable to other (especially robotic) domains, the models are designed for the robot soccer domain. Creating spatial agent models which have applicability across domains presents an interesting challenge. How can such models be structured so that reasoning with them is tractable? Are there spatial patterns which repeat across domains? What probabilistic representations are need to capture the types of agent behavior generally observed?

- **Using a Model of an Opponent in Complex Environments**

  Even if one has a model of an opponent, planning an appropriate response is a challenging problem. Section 6.3 presented one planning approach to responding to the predicted actions of an opponent. Using an opponent model in a standard discrete turn-taking game is a well known problem [e.g., Carmel and Markovitch, 1996]. However, it is challenging to use an opponent model in an environment like robot soccer where optimal policies are simply not known and continuous state and action spaces must be dealt with. One would like to employ regret minimizing techniques [e.g., Auer et al., 2002] to prevent being exploited while trying to exploit the opponent's weaknesses. However, it is not clear how such techniques could be applied in a domain as complex as robot soccer.

8.4.5 Coaching in General

While this thesis has helped define the coaching problem and identify the challenging research problems within, foundational and general coaching work still remains.

- **Formal Models of Coaching**

  A number of models of multi-agent systems have been proposed in recent years [Peshkin et al., 2000, Boutilier, 1996, Bernstein et al., 2000, Xuan et al., 2001, Pynadath and Tambe, 2002a,b]. However, none of these explicitly includes a coach agent. The models of teaching in the machine learning community (see Section 7.2) are all focused on classification problems, not action selection problems. Defining precisely the coach-agent interactions should allow for more reusability across domains as well as possibly suggesting solution algorithms. These models could also be useful in the design of future advice languages. While we have made some progress on this front (our preliminary models are described in Appendix D), more remains to be done.

- **The Challenges of Integration in a Coach Agent**

  One point noted in the introduction is that while many aspects of the coaching problem have been addressed and all have been tested in fully functioning systems, not all of the systems are integrated. For instance, the advice adaptation discussed in Chapter 3 is separate from the environment model learning discussed in Chapter 4. Often, integrating a system yields unexpected challenges and/or surprising benefits. I expect the full integration of a coach agent to be no different. As many of the components discussed throughout the thesis could feed into each other, interesting interactions will develop and will merit exploration. Further, as coaching is applied to more domains (such as those suggested in Section 1.4), additional challenges will surely arise.

8.5 Concluding Remarks

This thesis explores many aspects of the problem faced by an automated coach agent in generating advice for other agents. Machine learning is applied to learning about the environment, adversarial agents, and the advice receivers. We present a multi-agent plan representation to explore how a coach's advice could be used more effectively. I believe that as our agents live longer, become more complex, and use richer representations of the world, coaching relationships will become both more common and necessary. This thesis works to make those relationships as productive as possible.

Appendix A

CLang

This appendix describes CLang, the standard advice language for the simulated robot soccer environment (Section 2.2). Section 2.2.2, which provides a general overview of the language, should be read first. This language represents the collaborative effort of several people, including myself, on the RoboCup coach mailing list [Coach Mailing List].

Section A.1 will describe each of the elements of CLang, how they are composed, and what they mean. Section A.2 will provide several illustrative examples. Lastly, Section A.3 provides the full grammar for the language.

A.1 Description of CLang Elements

CLang has a number of different types of elements, each of which can be instantiated in different ways. This section discusses all of the elements and how they can be composed.

A.1.1 Points and Regions

Regions are used in both the conditions and actions of the language. Regions can be specified in a number of ways and any of the following forms can be used anywhere a region is expected. The names provided for the points and regions are the ones commonly used by the community. Examples of actual CLang for each will be given.

Regions are based on points. The types of points are shown in Table A.1. Anywhere a point is required, any of the types can be used. Note that some points refer to the current state of the world; the agent must evaluate this every time the point is used.

Name / Description	Example
Simple	`(pt 10 20)`
A global (x, y) location on the field	
Ball	`(pt ball)`
The global location of the ball	
Player	`(pt our 2)`
The global location of a player, either our team or any opponent. Note that the number can be a variable (more on variables in Section A.1.4).	
Arithmetic	`((pt ball) + (pt 5 10))`
The basic arithmetic operations $+$, $-$, $*$, and $/$ can be performed on points. Each operation works component-wise.	

Table A.1: The types of points in CLang

The list of regions is shown in Table A.2. The regions are various geometric shapes defined using the points above. Note that all points and regions use global coordinates rather than player specific coordinates. Therefore, CLang can not express concepts like "the area currently in the middle of your field of view." However, through use of player points, CLang can express regions like "the area between 10m and 20m further upfield than you."

A.1.2 Conditions

Given the definition of regions, we can now define the conditions of CLang. Conditions are constructed from logical connectives (and, or, not) of the atoms shown in Table A.3. The truth values of the atoms are based on the current state of the world.

One of the goals for choosing this set of conditions is that they are easily operationalizable by the agents. That is, the conditions refer to well-defined features of the world state. Conditions like "on offense" were rejected by the community because it was felt the interpretations of the agents would vary greatly, making the coach's task much harder. Note that the agents do not have a perfect view of the world, so they will naturally sometimes get the truth value of a condition wrong compared to the ground truth known by the server.

Name / Description	Example
Point	`(pt ball)`
Any point can be used as a region.	
Null	`(null)`
The empty region.	
Arc($p, r_i, r_o, \theta_b, \theta_s$)	`(arc (pt ball) 5 20 0 360)`
An arc region formed from part of a circle. Given a circle with center point p and radius r_o, this region is the part of the circle at least r_i (the inner radius) from p and whose angle to the center (in global coordinates) is between θ_b and $\theta_b + \theta_s$. The two angles θ_b and θ_s are known as the begin and span angles respectively.	
Triangle(p_1, p_2, p_3)	`(tri (pt -10 -10) (pt 0 0) (pt -20 20))`
A triangular region with vertices p_1, p_2, and p_3.	
Rectangle(p_1, p_2)	`(rec (pt ball) (pt 50 30))`
An axis-aligned rectangle with opposite corners p_1 and p_2.	
Union	`(reg (rec (pt 0 0) (pt ball)) (null))`
A union of other regions.	
Named	`"MyRegion"`
Any region can be assigned a name which can then be used instead of the region expression.	

Table A.2: The types of regions in CLang

Name Description	Example
True/False Constant truth values	`(true) (false)`
PlayerPos(s, c_{min}, c_{max}, r) Whether between c_{min} and c_{max} (inclusive) players from player set s are in region r. Note that variables representing player numbers could be in the player set.	`(ppos our {2 3 4} 1 2 "MyRegion")`
BallPos(r) Whether the ball is in the region r.	`(bpos "MyRegion")`
BallOwner(s) Whether a player in the player set s is currently the ball owner. The ball owner is defined as the last agent to touch the ball.	`(bowner opp {2 4 6 8 10})`
PlayMode(m) Whether the current play mode is m. Play modes represent the state of the game, such as "before kick off", "goal kick left", "play on", etc.	`(playm bko)`
GoalComparison An expression comparing the number of goals scored for or against to a given integer	`(our_goals >= 2)`
TimeComparison An expression about the current time of the game clock compared to a given integer	`(600 < time)`
UNum Used to restrict the possible values of a variable	`(unum X {1 2 3})`
Named Any condition can be assigned a name which can then be used instead of the expression.	`"MyCondition"`

Table A.3: The types of conditions in CLang

A.1.3 Actions

Similarly, the CLang actions were designed to have relatively clear semantics. CLang actions are recommended macro-actions for the agents and full list is given in Table A.4.

Two actions deserve special mention. For the "clear" action, there was confusion about whether this meant to clear "to" or "from" the region. Generally, the interpretation was at first "to" the region and was later changed to mean "from" the region.

"Home" is the least clearly defined action, but should be used to establish a formation for the team. Since "formation" means different things to different teams, the agents' interpretations vary. In general, the agent should be in or near the home region unless it has something else it is specifically doing.

Note that the exact meaning of a CLang action can change from step to step because the regions can change based on the state of the world. For example, the pass example in Table A.4 says to pass to any agent between 10m and 20m from the ball. Which agents satisfy this criteria will change throughout the game.

Also, the agents are given freedom in the implementation of the macro-actions in two primary ways. First, some actions have explicit choices, such as "mark *some* player in this set" or "position yourself at *some* point in this region." Secondly, the translation from these macro-actions to the lowest level actions is not fully specified. For example, neither how fast the ball should go for a pass nor the speed of dribbling is specified. This freedom allows for considerable variability in the implementations by the coachable agents.

A.1.4 Miscellany

One more component must be added to conditions and actions in order to make a rule. A directive contains: the list of actions, whether the actions are advised to be done or *not* be done, and which players the actions apply to. A rule is a condition combined with either a list of directives or a list of rules. Rules can be defined, redefined, turned on and off, and deleted. At every time step, the agents examine every currently turned on rule to see if the conditions match. If a turned-on rule (whose conditions match) refers to another rule, the latter rule is then checked (whether it is turned on or off).

Any region, condition, action, directive, or rule can be given a name which can be used in any place where one of those elements is expected. Because of a simple oversight, points can not be given names but this has not shown to be a major limitation.

As mentioned several times, CLang supports variables that represent player numbers.

Name Description	Example
Position(r) Position self in region r	`(pos (pt -10 10))`
Mark(s) Mark one of the players in player set s. Marking is a standard soccer term meaning to play defense against a player.	`(mark {7 8 9 10})`
MarkLine(s) Stay between the ball and one of the players in s	`(markl {5 6 7})`
MarkLine(r) Stay between the ball and the region r	`(markl (rec (pt 0 0) (pt 20 20)))`
Pass(r) Pass to a player in region r	`(pass (arc (pt ball) 10 20 0 360))`
Pass(s) Pass to a player in player set s	`(pass {0})`
Dribble(r) Dribble the ball to a point in region r	`(dribble "SomeRegion")`
Clear(r) Clear the ball to/from region r	`(clear "SomeOtherRegion")`
Shoot Shoot the ball	`(shoot)`
Hold Try to hold on to the ball, keeping it from opponents	`(hold)`
Intercept Try to get the ball	`(intercept)`
Tackle(s) Tackle (a basic soccer action) any opponent in player set s who has the ball	`(tackle {6 9})`
OffsidesLine(r) Move the offsides line (i.e. defensive position) up to region r	`(oline "MyName")`
HetType(t) Used for modeling rules to say that players are of heterogeneous type t. This action will not be used in this thesis.	`(htype 3)`
Home(r) Set the home region to r. See text for explanation.	`(home (rec (pt -30 -30) (pt -10 -10)))`
Named Any action can be assigned a name which can then be used instead of the expression	`"MyAction"`

Table A.4: The types of actions in CLang

Variables are one of the least understood and least well implemented parts of CLang, so this discussion represents my understanding of variables. The known limitations of current implementations will also be discussed.

Overall, the meaning of a rule with variables is that at each time step, for each assignment to all relevant variables such that the condition is true, the directive created by that assignment applies. Because of the flexibility allowed in the language through multiple uses of a variable in a condition and the use of variables in the points of regions, there is probably in general no more efficient way to evaluate an expression with variables than to try and match the condition for every possible assignment. Note that variables only bind to numbers, not to a complete player identification (a team and player number).

It is unknown exactly how complex a variable statement can be and still be matched correctly by the current coachable agents. However, expressions with one variable used in one location (that is *not* part of a region specification) are in general matched correctly. I believe that variables are not well integrated in CLang and hope that the community will work to revise the language in the future. Therefore, variables are used very sparingly throughout this thesis.

It was originally the intention that Clang could be used for both providing advice to the players and providing team and opponent modeling information. The same structure that means "If the world matches condition A, then do B" could be used to say "When the world looks like A, you (or your opponent) generally do B." However, at the time of writing, no coachable agent used any information provided through a modeling rule[1], so only advice directives are meaningful to agents.

A.2 Examples

In order to further clarify the type of concepts expressible by CLang, we provide a number of examples, illustrated and explained in detail. All of these examples are pulled from real utterances of our coach (though some names have been changed). In some cases the rules have been simplified slightly for illustration purposes.

In all cases, we illustrate the definition of a rule. In order for the players to actually use the rule, it would have to be turned on. If a rule named `Mary` was defined, it could be turned on like this:

[1] Actually, some agents can understand an "htype" action (see Table A.4), but this action is only for modeling rules and is qualitatively different from the other actions.

```
(rule (on Mary))
```

A.2.1 Example: Formation Based Marking

First, we consider a rule which uses two named CLang components in its definition. This example comes from the formation based marking described in Section 5.2.1

```
(definec "Andrade" (not (ppos our {1 3 4 5 6 7 8 9 10 11}
                        1 11 (arc (pt opp 11) 0 5 0 360))))
(defined "Baia" (do our {2} (mark {11})))
(definerule Deco direc
     ((and (playm play_on) "Andrade") "Baia"))
```

The first two lines define a condition (`definec` is the keyword introducing a condition definition) named "Andrade". Consider first the region defined beginning with the `arc` keyword. This region is illustrated in Figure A.1. Note that the center of the circle is wherever opponent number 11 is. The region includes all of the area between 0 and 5 meters from that player. The rest of the condition says, "if it is not the case that at least one player on our team (except player 2) is in that region." In other words, this condition tests whether "there are no teammates (except player 2) within 5 meters of opponent 11."

The next line defines a directive (introduced by the keyword `defined`) named "Baia." It advises player 2 on our team to mark player 11 on the opponent team. Finally, the rule named "Deco" is defined in the last line. It combines the named condition "Andrade" with the condition that the play mode is `play_on` (which is the normal play mode) and then uses the named directive "Baia." Overall, the rule says for player 2 to mark opponent 11, as long as no other player on the team is already marking that opponent.

A.2.2 Example: Setplays

Next, we will show the coach recommending positions for our team when the opponent has a goal kick.

```
(definerule Figo direc
        ((playm gk_opp)
                (do our {4} (pos (pt 36.5 -21.16)))
                (do our {10} (pos (pt 35 8)))
                (do our {11} (pos (pt 35 -9.16)))))
```

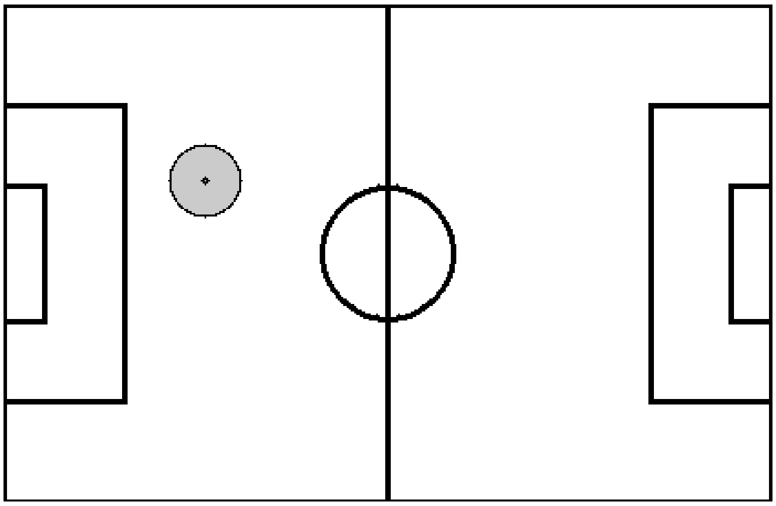


Figure A.1: Illustration of the region on the field used by the condition “Andrade.” Note that the center of the circular region is not a fixed point, but rather the current location of opponent 11.

The rule “Figo” has the condition that the play mode is the opponent’s goal kick. The rule then advises the players to position themselves in the regions illustrated in Figure A.2. Note that each of the regions is a single point.

As described in Chapter 6, the coach also makes plans for stopped ball situations when we control the ball. The first rule output by this process gives positions for some players as in the last example.

```
(definerule Pauleta direc
        ((or (playm bko) (playm ko_our) (playm ag_opp))
                (do our {4} (pos (pt 0 0)))
                (do our {9} (pos (pt -5 7)))
                (do our {10} (pos (pt -1 30.999)))))
```

These positions are illustrated in Figure A.3. The only additional complication in the rule is that several play mode conditions are combined with an `or`. The three play modes are “before kick off,” “our kick off,” and “after goal scored on opponent.” These three play modes are used in this example because there is some confusion among the coachable agents as to exactly what these play modes mean.

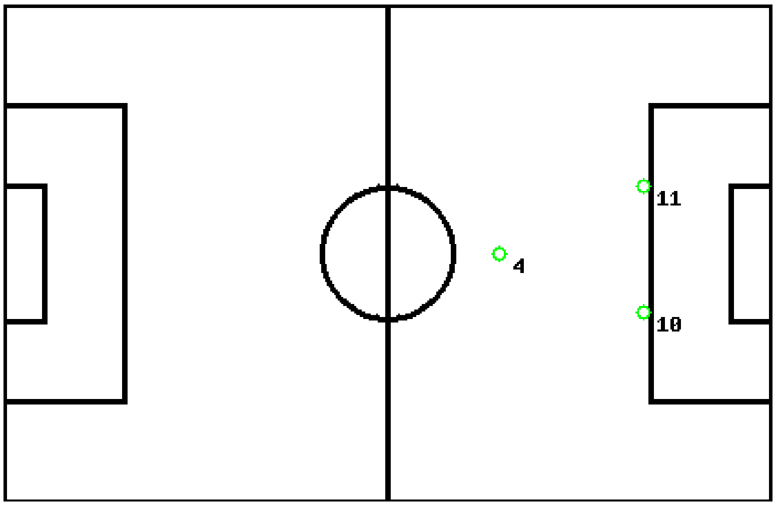


Figure A.2: Illustration of the positions advised by the "Figo" rule. The numbers refer to which player is supposed to stand at the point (represented as a small circle).

In order to actually move the ball for our kick off, the coach advises one agent to pass the ball.

```
(definerule Ronaldo direc
        ((time > 318) (do our {4} (pass {9}))))
```

The condition in the "Ronaldo" rule is that the time is larger than 318. Once that is true, player 4 is advised to pass to player 9 as instructed by the arrow in Figure A.3. Note that the real plans described in Chapter 6 would have more rules for the other steps in the plan.

A.2.3 Example: MDP Tree

We now move on to our final and most complicated example. This example comes from advice generated from a learned MDP as described in Chapter 4. The use of named regions and conditions will be combined with the use of nested rules.

First, a few regions and conditions are defined. The names will look unusual, but they encode somewhat how they will be used in the rest of the rules.

```
(definer "BG100658" (rec (pt 31.8 23) (pt 42.4 34.5)))
```

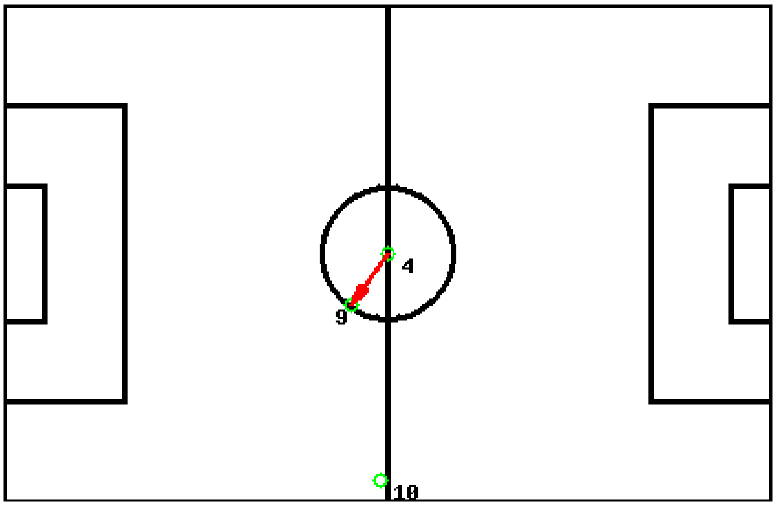


Figure A.3: Illustration of the "Pauleta" and "Ronaldo" rules. The "Pauleta" rule advises the positions shown by the circles for the given numbers. The "Ronaldo" rule specifies the pass from player 4 to player 9 shown by the arrow.

The region "BG100658" is depicted in Figure A.4. That region (and other similarly named ones) will be used in conditions on the position of the ball below.

```
(definer "PR4" (arc (pt ball) 1.5 15.05 15.1 52.52))
(definec "PC4_0" (and (ppos our {0} 0 0 "PR4")
                      (ppos opp {0} 0 0 "PR4")))
```

The region defined by "PR4" is shown in Figure A.5. This region (which has the ball at the center of the arc) is used in defining condition "PC4_0," which says that there are no teammates or opponents in region "PR4." Remember that the notation `{0}` for a player set means *all* players on the team.

We can now show some of the actual rules which use these regions. As mentioned previously, we will be using nested rules. We will define the innermost rule first.

```
(definerule A0000158 direc
  ((true)
    (do our {0} (pass (rec (pt 36 -20.16) (pt 52.5 20.16))))
    (do our {0} (hold))))
```

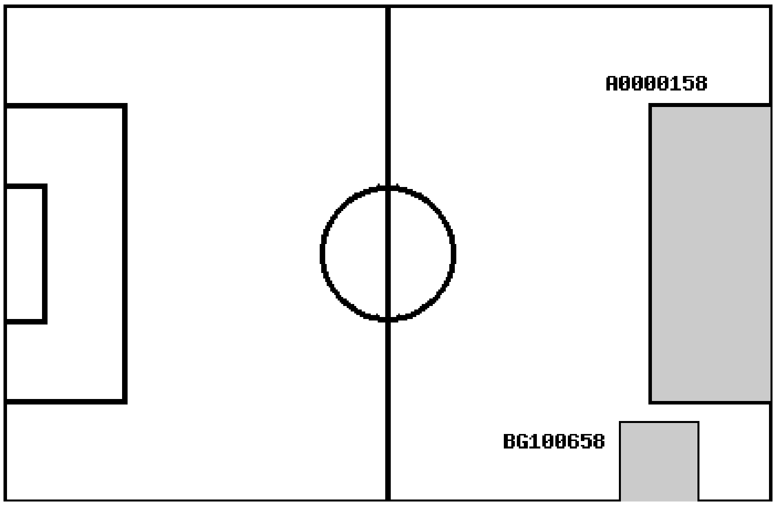


Figure A.4: Regions used in the nested rule example. One is the "BG100658" named region and the other is used in the rule "A0000158."

This rule named "A0000158" has just the condition `true` because the meaningful conditions will occur in the enclosing rules. The rule then recommends two actions to the entire team. The first is to pass to a player in the region illustrated in Figure A.4. The second recommendation is to hold on to the ball, keeping it away from opponents.

Now that this rule is defined, how is it used? Here is the rule which includes "A0000158."

```
(definerule A00001 direc
  ((true)
     ((bpos "BG100601") A0000101)
     ((bpos "BG100602") A0000102)
     ...
     ((bpos "BG100658") A0000158)))
```

The dots indicate that we have removed many similar statements. Rule "A00001" uses a form where every nested rule has its own condition. That is, overall the rule has the condition `true`, but in order for the agent to start trying to match the inner rule, the condition specified for that rule must match. In this case, all the conditions are on the position of the ball. If the ball is in region "BG100658" (shown in Figure A.4), then the rule "A0000158" will be checked to see if it matches.

In the same fashion, this rule can be nested further in other rules.

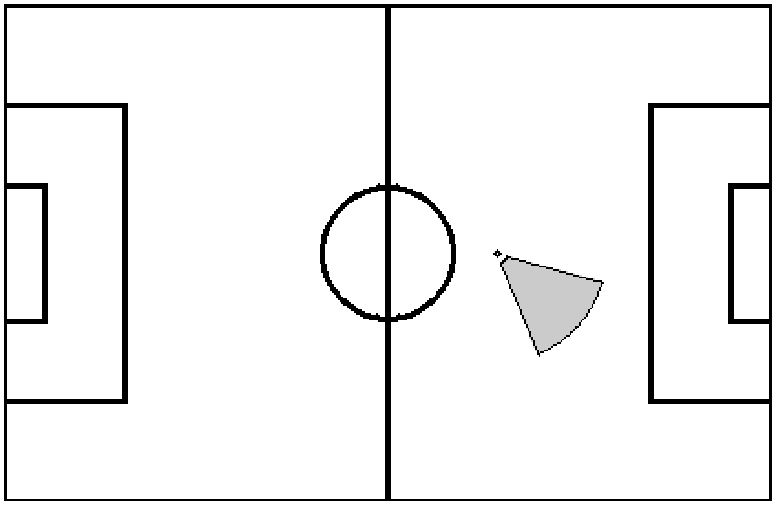


Figure A.5: The PR4 region used in the nested rule example. Note that the center of this arc region is the position of the ball.

```
(definerule A0000 direc
        ((true) ("PC4_0" A00000) ("PC4_1" A00001)))
(definerule A000 direc
        ((true) ("PC3_0" A0000) ("PC3_1" A0001)))
(definerule A00 direc
        ((true) ("PC2_0" A000) ("PC2_1" A001)))
(definerule A0 direc
        ((true) ("PC1_0" A00) ("PC1_1" A01)))
(definerule A direc
        ((true) ("PC0_0" A0) ("PC0_1" A1)))
```

We defined one condition “PC4_0” above and the other conditions have a similar structure. At the end of all of this, just the rule “A” would be turned on with:

```
(rule (on A))
```

Only the rules which are on are matched at the top level.

These examples naturally do not exhaustively cover the space of what can be expressed in CLang or even the space of what is expressed in this thesis. These examples do illustrate what CLang can express and how it is used.

A.3 CLang Grammar

What follows is the full grammar for CLang. Sections 2.2.2 and A.1 and the Soccer Server Manual [Chen et al., 2001] provide informal descriptions of the semantics. This text came from the Soccer Server Manual [Chen et al., 2001].

```
<MESSAGE> : <FREEFORM_MESS> | <DEFINE_MESS>
          | <RULE_MESS> | <DEL_MESS>

<RULE_MESS> : (rule <ACTIVATION_LIST>)

<DEL_MESS> : (delete <ID_LIST>)

<DEFINE_MESS> : (define <DEFINE_TOKEN_LIST>)

<FREEFORM_MESS> : (freeform <CLANG_STR>)

<DEFINE_TOKEN_LIST> : <DEFINE_TOKEN_LIST> <DEFINE_TOKEN>
                    | <DEFINE_TOKEN>

<DEFINE_TOKEN> :  (definec <CLANG_STR> <CONDITION>)
                | (defined <CLANG_STR> <DIRECTIVE>)
                | (definer <CLANG_STR> <REGION>)
                | (definea <CLANG_STR> <ACTION>)
                | (definerule <DEFINE_RULE>)

<DEFINE_RULE> :   <CLANG_VAR> model <RULE>
                | <CLANG_VAR> direc <RULE>

<RULE> : (<CONDITION> <DIRECTIVE_LIST>)
       | (<CONDITION> <RULE_LIST>)
       | <ID_LIST>

<ACTIVATION_LIST> : <ACTIVATION_LIST> <ACTIVATION_ELEMENT>
                  | <ACTIVATION_ELEMENT>

<ACTIVATION_ELEMENT> : (on|off <ID_LIST>)

<ACTION> : (pos <REGION>)
         | (home <REGION>)
```

```
          | (mark <UNUM_SET>)
          | (markl <UNUM_SET>)
          | (markl <REGION>)
          | (oline <REGION>)
          | (htype <INTEGER>)
          | <CLANG_STR>
          | (pass <REGION>)
          | (pass <UNUM_SET>)
          | (dribble <REGION>)
          | (clear <REGION>)
          | (shoot)
          | (hold)
          | (intercept)
          | (tackle <UNUM_SET>)

<CONDITION> : (true)
            | (false)
            | (ppos <TEAM> <UNUM_SET>
                    <INTEGER> <INTEGER> <REGION>)
            | (bpos <REGION>)
            | (bowner <TEAM> <UNUM_SET>)
            | (playm <PLAY_MODE>)
            | (and <CONDITION_LIST>)
            | (or <CONDITION_LIST>)
            | (not <CONDITION>)
            | <CLANG_STR>
            | (<COND_COMP>)
            | (unum <CLANG_VAR> <UNUM_SET>)
            | (unum <CLANG_STR> <UNUM_SET>)

<COND_COMP> : <TIME_COMP>
            | <OPP_GOAL_COMP>
            | <OUR_GOAL_COMP>
            | <GOAL_DIFF_COMP>

<TIME_COMP> : time <COMP> <INTEGER>
            | <INTEGER> <COMP> time

<OPP_GOAL_COMP> : opp_goals <COMP> <INTEGER>
                | <INTEGER> <COMP> opp_goals
```

```
<OUR_GOAL_COMP> : our_goals <COMP> <INTEGER>
                | <INTEGER> <COMP> our_goals

<GOAL_DIFF_COMP> : goal_diff <COMP> <INTEGER>
                 | <INTEGER> <COMP> goal_diff

<COMP> : < | <= | == | != | >= | >

<PLAY_MODE> : bko | time_over | play_on | ko_our | ko_opp
            | ki_our | ki_opp | fk_our | fk_opp
            | ck_our | ck_opp | gk_opp | gk_our
            | gc_our | gc_opp | ag_opp | ag_our


<DIRECTIVE> : (do|dont <TEAM> <UNUM_SET> <ACTION_LIST>)
            | <CLANG_STR>

<REGION> : (null)
         | (arc <POINT> <REAL> <REAL> <REAL> <REAL>)
         | (reg <REGION_LIST>)
         | <CLANG_STR>
         | <POINT>
         | (tri <POINT> <POINT> <POINT>)
         | (rec <POINT> <POINT>)

<POINT> : (pt <REAL> <REAL>)
        | (pt ball)
        | (pt <TEAM> <INTEGER>)
        | (pt <TEAM> <CLANG_VAR>)
        | (pt <TEAM> <CLANG_STR>)
        | (<POINT_ARITH>)

<POINT_ARITH> : <POINT_ARITH> <OP> <POINT_ARITH>
              | <POINT>

<OP> : + | - | * | /

<REGION_LIST> : <REGION_LIST> <REGION>
              | <REGION>

<UNUM_SET> : { <UNUM_LIST> }
```

```
<UNUM_LIST> : <UNUM>
            | <UNUM_LIST> <UNUM>

<UNUM> : <INTEGER> | <CLANG_VAR> | <CLANG_STR>

<ACTION_LIST> : <ACTION_LIST> <ACTION>
              | <ACTION>

<DIRECTIVE_LIST> : <DIRECTIVE_LIST> <DIRECTIVE>
                 | <DIRECTIVE>

<CONDITION_LIST> : <CONDITION_LIST> <CONDITION>
                 | <CONDITION>

<RULE_LIST> : <RULE_LIST> <RULE>
            | <RULE>

<ID_LIST> : <CLANG_VAR>
          | (<ID_LIST2>)
          | all

<ID_LIST2> : <ID_LIST2> <CLANG_VAR>
           | <CLANG_VAR>

<CLANG_STR> : "[0-9A-Za-z().+-*/?<>_ ]+"

<CLANG_VAR> : [abe-oqrt-zA-Z_]+[a-zA-Z0-9_]*

<TEAM> : our | opp
```

Appendix B

Transition Classification for MDP Learning

Learning an MDP as presented in Chapter 4 requires that abstract action templates be provided. The templates require a specification of the primary and secondary state transitions for each action. This appendix gives the details of the construction of the action class templates for the soccer and RCSSMaze environments. The construction follows the general scheme discussed in Section 4.3.3.

B.1 Simulated Robot Soccer

First, we have the list of features of pairs of states. In the names and descriptions below, we use past tense to refer to the first state and present tense to refer to the second state. Most features examine just one of the state factors (see Section 4.3.2).

Feature	**Factor(s)**	**Description**
IsMyGoal	Goal	In the our goal scored state
IsTheirGoal	Goal	In the their goal scored state
WasMyGoal	Goal	Were in the our goal scored state
WasTheirGoal	Goal	Were in the their goal scored state
WeHadBall	Ball Owner	We were the ball owner
TheyHadBall	Ball Owner	Their were the ball owner
WasFightBall	Ball Owner	No one/both teams were ball owners

Continued on next page

Feature	**Factor(s)**	**Description**
WeHaveBall	Ball Owner	We are the ball owner
TheyHaveBall	Ball Owner	They are the ball owner
IsFightBall	Ball Owner	No one/both teams are ball owners
WeKeptBall	Ball Owner	We were the ball owner and still are
TheyKeptBall	Ball Owner	They were the ball owner and still are
FightBall	Ball Owner	No one/both teams owner the ball in both states
MyFreeKickTran	Dead Ball	Was not our free kick and now it is
TheirFreeKickTran	Dead Ball	Was not their free kick and now it is
WasMyFreeKick	Dead Ball	Was our free kick
IsMyFreeKick	Dead Ball	Is our free kick
WasTheirFreeKick	Dead Ball	Was their free kick
IsTheirFreeKick	Dead Ball	Is their free kick
WasNotFreeKick	Dead Ball	Was not a free kick state
IsNotFreeKick	Dead Ball	Is not a free kick state
BallWasInMyPen	Ball Grid	The ball was in our penalty area
BallIsInMyPen	Ball Grid	The ball is in our penalty area
BallWasInTheirPen	Ball Grid	The ball was in their penalty area
BallIsInTheirPen	Ball Grid	The ball is in their penalty area
BallWasInMyHalf	Ball Grid	The ball was in my (defensive) half of the field
BallIsInMyHalf	Ball Grid	The ball is in my (defensive) half of the field
BallWasInTheirHalf	Ball Grid	The ball was in their half of the field
BallIsInTheirHalf	Ball Grid	The ball is in their half of the field
BallWasOnTopSide	Ball Grid	Ball was on the top of the field (negative y)
BallIsOnTopSide	Ball Grid	Ball is on the top of the field (negative y)
BallWasOnBottomSide	Ball Grid	Ball was on the bottom of the field (positive y)
BallIsOnBottomSide	Ball Grid	Ball is on the bottom of the field (positive y)
BallWasAtMyPenCorner	Ball Grid	Ball was at either corner of my penalty area. This is relevant because infractions in the penalty area result in the ball being moved here.
BallIsAtMyPenCorner	Ball Grid	*(see BallWasAtMyPenCorner, for current state)*
BallWasAtTheirPenCorner	Ball Grid	Ball was at either corner of their penalty area. This is relevant because infractions in the penalty area result in the ball being moved here.

Continued on next page

Feature	**Factor(s)**	**Description**
BallIsAtTheirPenCorner	Ball Grid	*(see BallWasAtTheirPenCorner, for current state)*
BallWasInMyTopCorner	Ball Grid	Ball was in the area directly above (negative y) my penalty area
BallIsInMyTopCorner	Ball Grid	Ball is in the area directly above (negative y) my penalty area
BallWasInMyBottomCorner	Ball Grid	Ball was in the area directly below (positive y) my penalty area
BallIsInMyBottomCorner	Ball Grid	Ball is in the area directly below (positive y) my penalty area
BallWasInTheirTopCorner	Ball Grid	Ball was in the area directly above (negative y) their penalty area
BallIsInTheirTopCorner	Ball Grid	Ball is in the area directly above (negative y) their penalty area
BallWasInTheirBottomCorner	Ball Grid	Ball was in the area directly below (positive y) their penalty area
BallIsInTheirBottomCorner	Ball Grid	Ball is in the area directly below (positive y) their penalty area
BallMoveOffensive	Ball Grid	The ball moved in the x direction towards towards my offensive side
BallMoveDefensive	Ball Grid	The ball moved in the x direction towards towards my defensive side
BallMoveDirNeutral	Ball Grid	The ball's x position did not change
BallMoveNone	Ball Grid	The ball is in the same place
BallMoveNear	Ball Grid	The ball moved to one of the 8 adjacent grid squares
BallMoveMid	Ball Grid	The ball moved to one of the 18 grid squares which are exactly 2 steps away in either the x or y dimension
BallMoveFar	Ball Grid	The ball moved to one of the 24 grid squares which are exactly 3 steps away in either the x or y dimension
BallMoveVeryFar	Ball Grid	All ball movements not covered by BallMoveNone, BallMoveNear, BallMoveMid, and BallMoveFar

Continued on next page

Feature	Factor(s)	Description
SelfTran	all	The states are the same
WeWereNotCloseToBall	Player Occupancy	Only meaningful for BallArc. No teammates were in the 2 regions closest to the ball
WeAreNotCloseToBall	Player Occupancy	Only meaningful for BallArc. No teammates are in the 2 regions closest to the ball
TheyWereNotCloseToBall	Player Occupancy	Only meaningful for BallArc. No opponents were in the 2 regions closest to the ball
TheyAreNotCloseToBall	Player Occupancy	Only meaningful for BallArc. No opponents are in the 2 regions closest to the ball

Based on these state features, transition classes can be defined. A transition class is defined by a DNF formula on state features. ! is used to negate literals. The expression “FeatureA FeatureB OR !FeatureC FeatureD” means that this class matches if FeatureA and FeatureB are true or if FeatureC is false and FeatureD is true. Remember that this list is ordered; every transition is assigned to the first matching class.

Class	Matching Expression
MyGoal	IsMyGoal WeHadBall
TheirGoal	IsTheirGoal TheyHadBall
MyFightGoal	IsMyGoal WasFightBall
TheirFightGoal	IsTheirGoal WasFightBall
MyOwnGoal	IsMyGoal TheyHadBall
TheirOwnGoal	IsTheirGoal WeHadBall
GoalToKickOff	WasMyGoal IsTheirFreeKick OR WasTheirGoal IsMyFreeKick
MyKickOutSide	WeHadBall TheirFreeKickTran BallIsOnTopSide OR WeHadBall TheirFreeKickTran BallIsOnBottomSide OR WasFightBall TheirFreeKickTran BallIsOnTopSide OR WasFightBall TheirFreeKickTran BallIsOnBottomSide
TheirKickOutSide	TheyHadBall MyFreeKickTran BallIsOnTopSide OR TheyHadBall MyFreeKickTran BallIsOnBottomSide OR WasFightBall MyFreeKickTran BallIsOnTopSide OR WasFightBall MyFreeKickTran BallIsOnBottomSide
MyShotCaughtOrToGK	WeHadBall BallIsInTheirPen TheirFreeKickTran
TheirShotCaughtOrToGK	TheyHadBall BallIsInMyPen MyFreeKickTran
MyFightShotCaughtOrToGK	WasFightBall BallIsInTheirPen TheirFreeKickTran
TheirFightShotCaughtOrToGK	WasFightBall BallIsInMyPen MyFreeKickTran

Continued on next page

Class	Matching Expression
MyOffOrFreeKickFault	WeHadBall BallIsInTheirHalf TheirFreeKickTran OR WasFightBall BallIsInTheirHalf TheirFreeKickTran
TheirOffOrFreeKickFault	TheyHadBall BallIsInMyHalf MyFreeKickTran OR WasFightBall BallIsInMyHalf MyFreeKickTran
MyIllegalBackPassToGoalie	WeHadBall BallWasInMyHalf TheirFreeKickTran BallIsAtMyPenCorner
TheirIllegalBackPassToGoalie	TheyHadBall BallWasInTheirHalf MyFreeKickTran BallIsAtTheirPenCorner
MyFightIllegalBackPassToGoalie	WasFightBall BallWasInMyHalf TheirFreeKickTran BallIsAtMyPenCorner
TheirFightIllegalBackPassToGoalie	WasFightBall BallWasInTheirHalf MyFreeKickTran BallIsAtTheirPenCorner
MyApparentIllegalBackPassToGoalie	TheyHadBall BallWasInMyHalf TheirFreeKickTran BallIsAtMyPenCorner
TheirApparentIllegalBackPassToGoalie	WeHadBall BallWasInTheirHalf MyFreeKickTran BallIsAtTheirPenCorner
MyLegalBackPassToGoalie	WeHadBall BallWasInMyHalf MyFreeKickTran BallIsInMyPen
TheirLegalBackPassToGoalie	TheyHadBall BallWasInTheirHalf TheirFreeKickTran BallIsInTheirPen
MyFreeKickFault	TheirFreeKickTran BallIsInMyHalf WeHadBall
TheirFreeKickFault	MyFreeKickTran BallIsInTheirHalf TheyHadBall
TheirApparentKickOutSide	MyFreeKickTran WeHadBall BallIsOnTopSide OR MyFreeKickTran WeHadBall BallIsOnBottomSide
MyApparentKickOutSide	TheirFreeKickTran TheyHadBall BallIsOnTopSide OR TheirFreeKickTran TheyHadBall BallIsOnBottomSide
MyApparentOffOrFreeKickFault	TheyHadBall TheirFreeKickTran
TheirApparentOffOrFreeKickFault	WeHadBall MyFreeKickTran
MyApparentFightToOffOrFreeKickFault	WasFightBall TheirFreeKickTran
TheirApparentFightToOffOrFreeKickFault	WasFightBall MyFreeKickTran
MyGoalieMove	WasMyFreeKick IsMyFreeKick BallWasInMyPen BallIsInMyPen

Continued on next page

Class	**Matching Expression**
TheirGoalieMove	WasTheirFreeKick IsTheirFreeKick BallWasInTheirPen BallIsInTheirPen
MyUnusualFreeKickToFreeKick	WasMyFreeKick IsMyFreeKick !BallMoveNone
TheirUnusualFreeKickToFreeKick	WasTheirFreeKick IsTheirFreeKick !BallMoveNone
MyFailedKickIn	WasMyFreeKick IsTheirFreeKick BallWasOnTopSide BallIsOnTopSide OR WasMyFreeKick IsTheirFreeKick BallWasOnBottomSide BallIsOnBottomSide
TheirFailedKickIn	WasTheirFreeKick IsMyFreeKick BallWasOnTopSide BallIsOnTopSide OR WasTheirFreeKick IsMyFreeKick BallWasOnBottomSide BallIsOnBottomSide
MyFreeKickToGoalieCatchorGK	WasMyFreeKick BallWasInTheirHalf IsTheirFreeKick BallIsInTheirPen
TheiFreeKickToGoalieCatchorGK	WasTheirFreeKick BallWasInMyHalf IsMyFreeKick BallIsInMyPen
MyFreeKickToOut	WasMyFreeKick IsTheirFreeKick BallIsOnTopSide OR WasMyFreeKick IsTheirFreeKick BallIsOnBottomSide
TheirFreeKickToOut	WasTheirFreeKick IsMyFreeKick BallIsOnTopSide OR WasTheirFreeKick IsMyFreeKick BallIsOnBottomSide
MyFreeKickToOff	WasMyFreeKick IsTheirFreeKick BallIsInTheirHalf
TheirFreeKickToOff	WasTheirFreeKick IsMyFreeKick BallIsInMyHalf
MyGoalKickFault	WasMyFreeKick BallWasInMyPen IsTheirFreeKick BallIsAtMyPenCorner
TheirGoalKickFault	WasTheirFreeKick BallWasInTheirPen IsMyFreeKick BallIsAtTheirPenCorner
MyFreeKickToBackPass	WasMyFreeKick BallWasInMyHalf IsTheirFreeKick BallIsAtMyPenCorner
TheirFreeKickToBackPass	WasTheirFreeKick BallIsInTheirHalf IsMyFreeKick BallIsAtTheirPenCorner
MyFreeKickStart	WasMyFreeKick IsNotFreeKick
TheirFreeKickStart	WasTheirFreeKick IsNotFreeKick
MyCross	!TheyHadBall BallWasInTheirTopCorner BallIsInTheirPen !TheyHaveBall !BallMoveNone OR !TheyHadBall BallWasInTheirBottomCorner BallIsInTheirPen !TheyHaveBall !BallMoveNone

Continued on next page

Class	Matching Expression
MyFailedCross	WeHadBall BallWasInTheirTopCorner BallIsInTheirPen TheyHaveBall !BallMoveNone OR WeHadBall BallWasInTheirBottomCorner BallIsInTheirPen TheyHaveBall !BallMoveNone
MyPossFailedCross	WasFightBall BallWasInTheirTopCorner BallIsInTheirPen TheyHaveBall !BallMoveNone OR WasFightBall BallWasInTheirBottomCorner BallIsInTheirPen TheyHaveBall !BallMoveNone
TheirCross	!WeHadBall BallWasInMyTopCorner BallIsInMyPen !WeHaveBall !BallMoveNone OR !WeHadBall BallWasInMyBottomCorner BallIsInMyPen !WeHaveBall !BallMoveNone
TheirFailedCross	TheyHadBall BallWasInMyTopCorner BallIsInMyPen WeHaveBall !BallMoveNone OR TheyHadBall BallWasInMyBottomCorner BallIsInMyPen WeHaveBall !BallMoveNone
TheirPossFailedCross	WasFightBall BallWasInMyTopCorner BallIsInMyPen WeHaveBall !BallMoveNone OR WasFightBall BallWasInMyBottomCorner BallIsInMyPen WeHaveBall !BallMoveNone
MyVeryLongPassOrClear	!TheyHadBall BallMoveVeryFar BallMoveOffensive WeHaveBall
TheirVeryLongPassOrClear	!WeHadBall BallMoveVeryFar BallMoveDefensive TheyHaveBall
MyVeryLongSidePass	!TheyHadBall BallMoveVeryFar BallMoveDirNeutral WeHaveBall
TheirVeryLongSidePass	!WeHadBall BallMoveVeryFar BallMoveDirNeutral TheyHaveBall
MyFailedVeryLongPassOrClear	!TheyHadBall BallMoveVeryFar BallMoveOffensive !WeHaveBall
TheirFailedVeryLongPassOrClear	!WeHadBall BallMoveVeryFar BallMoveDefensive !TheyHaveBall
MyFailedVeryLongSidePass	!TheyHadBall BallMoveVeryFar BallMoveDirNeutral !WeHaveBall
TheirFailedVeryLongSidePass	!WeHadBall BallMoveVeryFar BallMoveDirNeutral !TheyHaveBall

Continued on next page

Class	**Matching Expression**
MyVeryLongBackPass	!TheyHadBall BallMoveVeryFar WeHaveBall
TheirVeryLongBackPass	!WeHadBall BallMoveVeryFar TheyHaveBall
MyFailedVeryLongBackPass	!TheyHadBall BallMoveVeryFar !WeHaveBall
TheirFailedVeryLongBackPass	!WeHadBall BallMoveVeryFar !TheyHaveBall
MyLongPassOrClear	!TheyHadBall BallMoveFar BallMoveOffensive WeHaveBall
TheirLongPassOrClear	!WeHadBall BallMoveFar BallMoveDefensive TheyHaveBall
MyLongSidePass	!TheyHadBall BallMoveFar BallMoveDirNeutral WeHaveBall
TheirLongSidePass	!WeHadBall BallMoveFar BallMoveDirNeutral TheyHaveBall
MyFailedLongPassOrClear	!TheyHadBall BallMoveFar BallMoveOffensive !WeHaveBall
TheirFailedLongPassOrClear	!WeHadBall BallMoveFar BallMoveDefensive !TheyHaveBall
MyFailedLongSidePass	!TheyHadBall BallMoveFar BallMoveDirNeutral !WeHaveBall
TheirFailedLongSidePass	!WeHadBall BallMoveFar BallMoveDirNeutral !TheyHaveBall
MyLongBackPass	!TheyHadBall BallMoveFar WeHaveBall
TheirLongBackPass	!WeHadBall BallMoveFar TheyHaveBall
MyFailedLongBackPass	!TheyHadBall BallMoveFar !WeHaveBall
TheirFailedLongBackPass	!WeHadBall BallMoveFar !TheyHaveBall
MyMidPassOrClear	!TheyHadBall BallMoveMid BallMoveOffensive WeHaveBall
MyFailedMidPassOrClear	!TheyHadBall BallMoveMid BallMoveOffensive !WeHaveBall
MyMidSidePassOrClear	!TheyHadBall BallMoveMid BallMoveDirNeutral WeHaveBall
MyFailedMidSidePassOrClear	!TheyHadBall BallMoveMid BallMoveDirNeutral !WeHaveBall
MyMidBackPass	!TheyHadBall BallMoveMid WeHaveBall
MyFailedMidBackPass	!TheyHadBall BallMoveMid !WeHaveBall
TheirMidPassOrClear	!WeHadBall BallMoveMid BallMoveDefensive TheyHaveBall

Continued on next page

Class	Matching Expression
TheirFailedMidPassOrClear	!WeHadBall BallMoveMid BallMoveDefensive !TheyHaveBall
TheirMidSidePassOrClear	!WeHadBall BallMoveMid BallMoveDirNeutral TheyHaveBall
TheirFailedMidSidePassOrClear	!WeHadBall BallMoveMid BallMoveDirNeutral !TheyHaveBall
TheirMidBackPass	!WeHadBall BallMoveMid TheyHaveBall
TheirFailedMidBackPass	!WeHadBall BallMoveMid !TheyHaveBall
MyShortPassOrDribble	!TheyHadBall BallMoveNear WeHaveBall
MyFailedShortPassOrDribble	!TheyHadBall BallMoveNear !WeHaveBall
TheirShortPassOrDribble	!WeHadBall BallMoveNear TheyHaveBall
TheirFailedShortPassOrDribble	!WeHadBall BallMoveNear !TheyHaveBall
MyLostBall	WeHadBall BallMoveNone !TheyWereNotCloseToBall TheyHaveBall
TheirLostBall	TheyHadBall BallMoveNone !WeWereNotCloseToBall WeHaveBall
MyLostBallToFight	WeHadBall BallMoveNone !TheyWereNotCloseToBall IsFightBall
TheirLostBallToFight	TheyHadBall BallMoveNone !WeWereNotCloseToBall IsFightBall
MyWonFight	WasFightBall BallMoveNone WeHaveBall
TheirWonFight	WasFightBall BallMoveNone TheyHaveBall
ContinueFightSelfTran	WasFightBall SelfTran
ContinueFight	WasFightBall BallMoveNone IsFightBall
MyFailedClosePass	WeHadBall BallMoveNone TheyWereNotCloseToBall TheyHaveBall
TheirFailedClosePass	TheyHadBall BallMoveNone WeWereNotCloseToBall WeHaveBall
MyPassToFightClose	WeHadBall BallMoveNone TheyWereNotCloseToBall IsFightBall
TheirPassToFightClose	TheyHadBall BallMoveNone WeWereNotCloseToBall IsFightBall
MyHoldBallSelfTran	WeHadBall SelfTran
TheirHoldBallSelfTran	TheyHadBall SelfTran
MyHoldBall	WeHadBall BallMoveNone WeHaveBall

Continued on next page

Class	Matching Expression
TheirHoldBall	TheyHadBall BallMoveNone TheyHaveBall

Finally, we can define the classes that make up the primary and secondary transitions.

Action	Primary Transition Classes	Secondary Transition Classes
HoldBall	MyHoldBall MyHoldBallSelfTran MyWonFight MyWonFightSelfTran ContinueFightSelfTran ContinueFight MyLostBall MyLostBallToFight	
PassTo	MyVeryLongPassOrClear MyVeryLongSidePass MyVeryLongBackPass MyLongPassOrClear MyLongSidePass MyLongBackPass MyMidPassOrClear MyMidSidePassOrClear MyMidBackPass MyShortPassOrDribble MyIllegalBackPassToGoalie MyFightIllegalBackPassToGoalie	MyOffOrFreeKickFault MyFailedVeryLongPassOrClear MyFailedVeryLongSidePass MyFailedVeryLongBackPass MyFailedLongPassOrClear MyFailedLongSidePass MyFailedLongBackPass MyFailedMidPassOrClear MyFailedMidSidePassOrClear MyFailedMidBackPass MyFailedShortPassOrDribble MyLostBall MyFailedClosePass MyPassToFightClose
DribbleTo	MyShortPassOrDribble	MyFailedShortPassOrDribble MyLostBall MyLostBallToFight
Clear	MyKickOutSide MyVeryLongPassOrClear MyVeryLongSidePass MyLongPassOrClear MyLongSidePass MyMidPassOrClear MyMidSidePassOrClear MyFailedVeryLongPassOrClear MyFailedVeryLongSidePass MyFailedLongPassOrClear MyFailedLongSidePass MyFailedMidPassOrClear MyFailedMidSidePassOrClear MyLostBall MyFailedClosePass MyPassToFightClose	

Continued on next page

Action	Primary Transition Classes	Secondary Transition Classes
Shoot	MyGoal MyFightGoal MyShotCaughtOrToGK MyFightShotCaughtOrToGK MyOffOrFreeKickFault MyLostBall	
Cross	MyCross MyFailedCross MyPossFailedCross	
TheirAct	*(all beginning with "Their" except those in Null)*	
Null	MyOwnGoal MyGoalieMove MyFreeKickStart MyUnusualFreeKickToFreeKick MyLegalBackPassToGoalie MyFreeKickFault MyApparentKickOutSide MyApparentOffOrFreeKickFault MyApparentFightToOffOrFreeKickFault MyApparentIllegalBackPassToGoalie TheirApparentIllegalBackPassToGoalie GoalToKickOff MyFailedKickIn MyFreeKickToGoalieCatchorGK MyFreeKickToOff MyFreeKickToOut MyGoalKickFault MyFreeKickToBackPass TheirApparentKickOutSide TheirApparentOffOrFreeKickFault TheirApparentFightToOffOrFreeKickFault	

B.2 RCSSMaze

The action template definition is simpler for RCSSMaze than for soccer. First we have we have the features of the abstract state pairs.

Feature	Description
WasGoal	We were at the goal state
IsGoal	We are at the goal state
WasBegin	We were at the initial state
IsBegin	We are at the initial state
HadBall	The agent (or both agent and wall) had the ball
HasBall	The agent (or both agent and wall) has the ball
WallHadBall	The wall (or both agent and wall) had the ball
WallHasBall	The wall (or both agent and wall) has the ball
BallIsInMyPen	*(same as soccer environment above)*

Continued on next page

Feature	Description
BallMoveNone	*(same as soccer environment above)*
BallMoveNear	*(same as soccer environment above)*
BallMoveMid	*(same as soccer environment above)*
BallMoveFar	*(same as soccer environment above)*
BallMoveVeryFar	*(same as soccer environment above)*

Next, we have the transition classes.

Class	Matching Expression
ToGoal	!WasGoal IsGoal
GoalSelfTran	WasGoal IsGoal
GoalReset	WasGoal IsBegin
BeginSelfTran	WasBegin IsBegin
WallToBegin	WallHadBall !HadBall !WasBegin IsBegin
FightToBegin	WallHadBall HadBall !WasBegin IsBegin
DribbleToBegin	!WallHadBall HadBall IsBegin
GotBallFromBegin	WasBegin HasBall !WallHasBall
BeginToWall	WasBegin WallHasBall
WallToWall	!HadBall WallHadBall !HasBall WallHasBall
WallToFight	!HadBall WallHadBall HasBall WallHasBall
WonFight	HadBall WallHadBall HasBall !WallHasBall
LostFight	HadBall WallHadBall !HasBall WallHasBall
ToFight	HadBall !WallHadBall HasBall WallHasBall BallMoveNone
DribbleToFight	HadBall !WallHadBall HasBall WallHasBall BallMoveNear
LostToWall	HadBall !WallHadBall !HasBall WallHasBall BallMoveNone
DribbleLostToWall	HadBall !WallHadBall !HasBall WallHasBall BallMoveNear
WallGaveToAgent	!HadBall WallHadBall HasBall !WallHasBall
StationaryFightContinue	HadBall WallHadBall HasBall WallHasBall BallMoveNone
MobileFightContinue	HadBall WallHadBall HasBall WallHasBall BallMoveNear
WallToNearBegin	!HadBall WallHadBall HasBall !WallHasBall BallIsInMyPen
MidKickToWall	HadBall !WallHadBall !HasBall WallHasBall BallMoveMid
FarKickToWall	HadBall !WallHadBall !HasBall WallHasBall BallMoveFar
MidKickToFight	HadBall !WallHadBall HasBall WallHasBall BallMoveMid
FarKickToFight	HadBall !WallHadBall HasBall WallHasBall BallMoveFar
OtherAgentLost	!WallHadBall WallHasBall
HoldBall	HadBall HasBall BallMoveNone

Continued on next page

Class	Matching Expression
DribbleBall	HadBall HasBall BallMoveNear
MidDribbleBall	HadBall HasBall BallMoveMid
FarDribbleBall	HadBall HasBall BallMoveFar

Finally, we combine these classes to define the action templates. Since we are still using CLang, the actions are quite similar to the soccer actions.

Action	Primary Transition Classes	Secondary Transition Classes
HoldBall	HoldBall ToFight LostToWall	
DribbleTo	GotBallFromBegin BeginToWall DribbleToFight DribbleLostToWall DribbleBall MidDribbleBall FarDribbleBall MidKickToWall FarKickToWall MidKickToFight FarKickToFight	HoldBall ToFight LostToWall DribbleToFight DribbleLostToWall MidKickToWall FarKickToWall MidKickToFight FarKickToFight
Shoot	ToGoal	
TheirAct	WallToBegin FightToBegin WallToWall WallToFight WonFight LostFight WallGaveToAgent StationaryFightContinue MobileFightContinue WallToNearBegin	
Null	GoalSelfTran GoalReset BeginSelfTran DribbleToBegin OtherAgentLost	

Appendix C

Source Code and Data Files

This thesis has been heavily experimental and has consequently produced a large amount of data. Almost all source code used to run these the experiments is available at the thesis web page `http://www.cs.cmu.edu/~pfr/thesis/`. Since the complete data files are about 94GB, they may not be available on the web page, but will be by request.

The web page also gives a more detailed list of the experimental data from this thesis. Most, but not all, of the experiments are discussed in the main text. Roughly, the data breaks down into several categories:

- Experiments in predator-prey environment (see Chapter 3)
- Experiments recreating the coach competition from RoboCup 2001 (see Section 5.2.3)
- Experiments after RoboCup 2003 and 2004, focused on (but not exclusively) the MDP learning (see Chapter 4)
- Experiments in the RCSSMaze environment (see Section 4.4.3)
- Data from the MASTN experiments (see Chapter 6)

Appendix D

Beginnings of a Formal Model of Coaching

One of the goals of this thesis is to better define the coaching problem. As part of this work, we developed some preliminary descriptive formal models of the coaching problem. These models draw from other models of multi agent systems such as MMDP [Boutilier, 1999] and COM-MTDP [Pynadath and Tambe, 2002a].

These models are still somewhat preliminary. We believe that they do have descriptive power, but we are not yet able to use these models to reason much about the coaching problem.

D.1 Direct, General Model

This model is an extension of a single-agent MDP to include a coach. The world is fully observable by both the coach and the agent, and the coach is coaching a single agent. The communication is one way, with the coach talking to the agent, but not vice versa. The coach's only action is to communicate to the agent, and the coach does not directly affect the transitions or rewards of the world.

Essentially no restriction is put on the format or meaning of the language between the coach and agent.

Figure D.1 graphically overviews the model, showing how information will flow around. The details of all the components are described below.

Our model is a triple of tuples. The first primarily describes the agent, the second de-

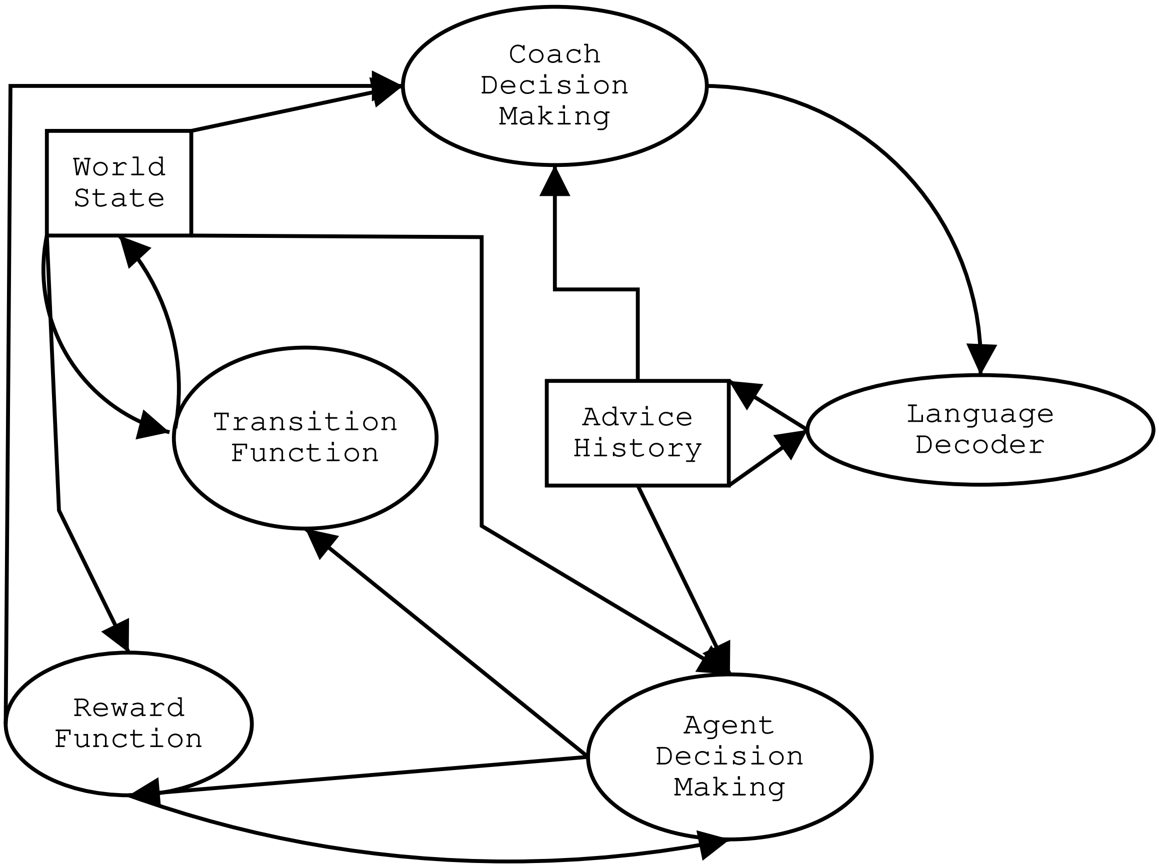


Figure D.1: Graphical description of coach agent interaction in most general model scribes the coach, and the third is shared information or attributes: $\langle\langle \mathcal{S}, \mathcal{A}\rangle, \langle \mathcal{L}, \Delta\rangle, \langle T, R, \mathcal{Y}, Y, y_0\rangle\rangle$

$\mathcal{S}$ The set of states

$\mathcal{A}$ The set of actions *for the agent*

$\mathcal{L}$ The language set, the set of all things that a coach could say

Δ Gives the interval for advice giving for the coach. The meaning is that every Δ steps, the coach gets to say something from $\mathcal{L}$.

$T\colon\ \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ The transition function. Given a state of the world and an agent action, gives a probability distribution over next states.

$R\colon\ \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ The reward function

$\mathcal{Y}$ The set of advice summary states. The advice summary state is designed to summarize in some fashion the entire history of communication from coach to agent (elements of $\mathcal{L}$). In general, this could be the entire communication history. However, the summary state could be used to represent such commands as "forget this earlier piece of advice."

$Y\colon\ \mathcal{Y} \times \mathcal{L} \times \mathcal{S} \to \mathcal{Y}$ The language to advice summary interpretation function. This function is used to update the advice summary state given what the coach said (an element of $\mathcal{L}$) and the current state (an element of $\mathcal{S}$). Note in particular that the advice can be situated. That is, the current state when the advice is received can affect the interpretation of the advice.

$y_0 \in \mathcal{Y}$ The initial advice summary state.

Given these components, we can now describe fully how the world operates. The agent and the coach both know the language to advice summary function Y and the current advice summary state $y_{t-1} \in \mathcal{Y}$. We are in a state of the world ($s_t \in \mathcal{S}$). If this is the first time step or it has been Δ steps since the last time the coach talked, the coach uses y_{t-1} and s_t to pick something to say ($l_t \in \mathcal{L}$). Note that this means the the coach's policy is $\pi_c\colon\ \mathcal{Y} \times \mathcal{S}^{\Delta} \to \mathcal{L}$ (the coach explicitly remembers the last Δ states). If the coach said something, the language to advice summary function is used to compute the new summary state $y_t = Y(y_{t-1}, l_t, s_t)$ (both the coach and the agent can do this computation without explicitly sharing anything but l_t). If the coach did not say anything, then $y_t = y_{t-1}$. The agent then uses y_t and s_t to pick an action from $\mathcal{A}$. Note that this means the agent's policy is $\pi_a\colon\ \mathcal{Y} \times \mathcal{S} \to \mathcal{A}$.

The agent's action determines a state transition (s_{i+1}) and reward (r_t) as in a normal MDP. Both the agent and coach then receive reward r_t.

To summarize, the history of the world here can be represented as a sequence of tuples $\langle s_t, l_t, y_t, a_t, r_t \rangle$ where

s_t The state at time t

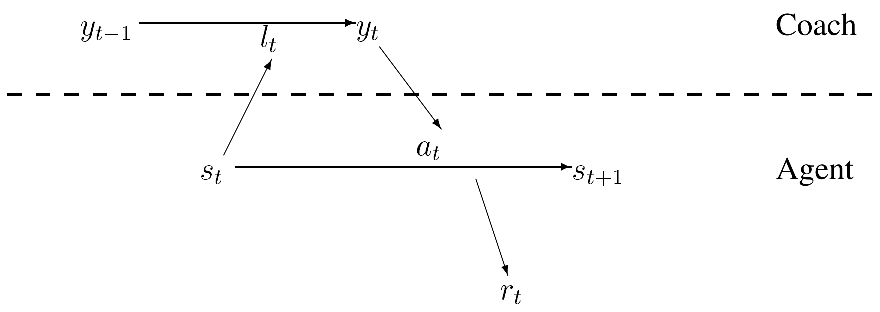
l_t What the coach says at time t. This can be $\emptyset$ to represent that nothing was said.

y_t The advice summary state at time t (updated to include l_t)

a_t The agent's action at time t

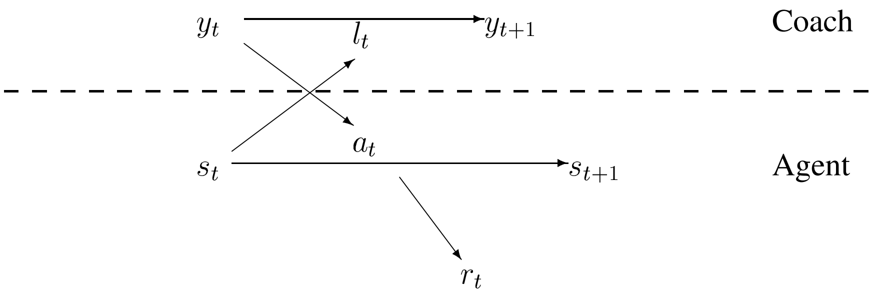
r_t The reward received at time t

Alternatively, this can be summarized with a diagram describing how the history variables evolve over time



The separation between the coach and agent is meant to signify what variables the coach and agent influence; the coach controls updates to the advice summary state y_t through its choice of what to say l_t. The agent chooses actions a_t to update the state s_t.

Note that in the model described thus far, the coach is able to see the state and say something *before* the agent has to choose an action. A slight variation is to make the coach say something and the agent act simultaneously. That is, when the agent is in a given state, it does not get the hear the coach advice before having to decide on an action. This is pictured here:



D.1.1 Examples

This section contains instantiates the model to match some coaching environments we have worked with.

Predator-Prey Continual Advice

This instantiates the model to match the predator-prey continual advice model as described in Section 3.1.

$\mathcal{S}$ Location of the predator and preys on the grid world.

$\mathcal{A}$ {North, South, East, West, Stay}

$T\colon\ \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ This is the obvious thing with the prey moving randomly.

$R\colon\ \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ The action is ignored and any state where the prey is captured is rewarded as described in the predator-prey paper.

$\mathcal{L}$ This is just $\mathcal{A}$; the coach recommends actions.

$\mathcal{Y}$ This is just $\mathcal{A}$; the last recommendation the coach made is the only thing remembered.

$Y\colon\ \mathcal{Y} \times \mathcal{L} \times \mathcal{S} \to \mathcal{Y}$ The advice summary state is just the last thing that the coach said: $Y(y_{t-1}, l_t, s_t) = l_t$

$y_0 \in \mathcal{Y}$ It doesn't matter what this is as it is never used.

Δ is 1; the coach can talk every step.

Predator-Prey Limited Bandwidth

This instantiates the model to match the predator-prey with limited bandwidth as described in Section 3.3.

$\mathcal{S}$ Location of the predator and preys on the grid world (as in previous).

$\mathcal{A}$ $\{$North, South, East, West, Stay$\}$

$T\colon\ \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ This is the obvious thing (prey move randomly).

$R\colon\ \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ The action is ignored and any state where the prey is captured is rewarded as described in the predator-prey paper (same as before).

$\mathcal{L}$ The set is $\langle \mathcal{S}, \mathcal{A} \rangle^K$. That is, a sequence of K (the parameter controlling the bandwidth) state-action pairs.

$\mathcal{Y}$ The set of functions $\mathcal{S} \to \mathcal{A} \cup \{\emptyset\}$. This simply "remembers" one recommended action for each state, or the null indicating that no advice has been received.

$Y\colon\ \mathcal{Y} \times \mathcal{L} \times \mathcal{S} \to \mathcal{Y}$ The sequence advice replaces the stored recommended action in the natural way.

$y_0 \in \mathcal{Y}$ Initialize to all nulls, $\forall s \in \mathcal{S}, y_0(s) = \emptyset$

Δ is I, the interval when the coach can talk.

This is a bit trickier as we had to use the Δ parameter and have the agent remember past advice.

D.2 A More Specific Model

While the model in Section D.1 does describe some coaching environments as shown in Section D.1.1, it is not as useful as we would like. We want to use the model to develop coaching algorithms that apply across domains. Note especially that in the previous model there is a complete lack of commitment to the structure of the language set $\mathcal{L}$, the advice summary set $\mathcal{Y}$, and, consequently, the structure of the language to advice summary function Y.

In this section, we present a more specific model which imposes more structure. This is a strict specialization; all models of this type are models of the more general type.

The basic idea is to require that the language provides (sets of) goal states to the agent. That is, the coach is giving the agent a goal, which is represented as a subset of the state space with the meaning of "Please get to one of the states in this subset." Over time, the agent could be given several possible goals for a given state.

Our model is a triple of tuples. The first tuple is for the agent, the second if for the coach, and the third is shared information: $\langle\langle \mathcal{S}, \mathcal{A}\rangle, \langle \mathcal{L}, \Delta\rangle, \langle T, R, L\rangle\rangle$

$\mathcal{S}$ The set of states (same as an MDP)

$\mathcal{A}$ The set of actions *for the agent* (same as an MDP)

$\mathcal{L}$ The language set, the set of all things that a coach could say. This is the same as before; no constraints on the format.

Δ The interval at which the coach can talk (same as before).

$T\colon\ \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ The transition function. Given a state of the world and an agent action, gives a probability distribution over next states (same as an MDP).

$R\colon\ \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ The reward function (same as an MDP)

L The language decoder function. This is a new part of the model. This function translates elements of $\mathcal{L}$ into goal states for particular states.

$$L\colon\ \mathcal{S} \times \mathcal{L} \to ((\mathcal{S} \to \mathcal{P}(\mathcal{S})) \times (\mathcal{S} \to \mathcal{P}(\mathcal{S}))) \tag{D.1}$$

In order words, each element of $\mathcal{L}$ gives two functions on the state space. The first means "For each of these states, here is a goal" and the second means "You should no longer try to obtain this goal." If the goal is the empty set, then that element of the language says nothing about that state.

The advice is situated. The meaning of a piece of advice can change depending on what state the agent is in when it hears it.

Since we claimed this is a strict specialization of the previous model, the reader may be asking where the missing elements of the tuple are. We can define those in terms of the given elements

$\mathcal{Y}$ The set of advice summary states. This stores, for each state, a set of goals, where each goal is represented as a subset of the state space.

$$\mathcal{Y} := \{y \mid y\colon\ \mathcal{S} \to \mathcal{P}(\mathcal{P}(\mathcal{S}))\} \tag{D.2}$$

$Y\colon\ \mathcal{Y} \times \mathcal{L} \times \mathcal{S} \to \mathcal{Y}$ The language to advice summary interpretation function. In order to write this down precisely, we are going to define two new operations on functions to make things simpler (yes, believe it or not, this is simpler).

First let's define $X \oplus Y$. Intuitively, this is a function set addition; add the output of one function the output of a second function. The input types are (for some sets $\mathcal{A}$ and $\mathcal{B}$) $X\colon\ \mathcal{A} \to \mathcal{P}(\mathcal{B})$ and $Y\colon\ \mathcal{A} \to \mathcal{B}$. The return is of type $\mathcal{A} \to \mathcal{P}(\mathcal{B})$. Then, $\forall z \in \mathcal{A}$:

$$(X \oplus Y)(z) = \begin{cases} X(z) \cup \{Y(z)\} & \text{if } Y(z) \neq \emptyset \\ X(z) & \text{o.w} \end{cases} \tag{D.3}$$

The short story is that the output of the functions X and Y are merged; the new functions returns, for each input, everything that X and Y did.

Now we define $X \ominus Y$, Intuitively, this is a function set subtraction; remove the output of one function from output of a second function. This is the same as above except:

$$(X \ominus Y)(z) = \begin{cases} X(z) \setminus \{Y(z)\} & \text{if } Y(z) \neq \emptyset \\ X(z) & \text{o.w} \end{cases} \tag{D.4}$$

Basically, instead of adding the output, the new function returns, for each input, everything that X did that Y did not.

Now we will define Y. Given the previous summary state y_{t-1}, the current coach utterance l_t, and the current state s_t, we will describe the next summary state y_t (the return of Y), using the new $\oplus$ and $\ominus$ operators, (with $\mathcal{A} = \mathcal{S}$ and $\mathcal{B} = \mathcal{P}(\mathcal{S})$). A is the function describing what goals to add and D is the function describing what goals to delete.

$$\langle A, D\rangle := L(s_t, l_t) \tag{D.5}$$

$$y_t = (y_{t-1} \oplus A) \ominus D \tag{D.6}$$

$y_0 \in \mathcal{Y}$ The initial advice summary state. $\forall s \in \mathcal{S}, y_0(s) = \emptyset$

The math here is not very pretty, but the idea is not that complicated. The coach is advising goals to the agent. A goal is simply a subset of states, with the meaning of "I advise getting to one of those states." The agent is simply remembering (with the set $\mathcal{Y}$) what goals the coach has provided for each state. The language decoder translates each language utterance into goals to add and remove for various states of the world.

Several final points should be made about this model:

- How/how long does the agent remember what was advised by the coach in previous states?
- We still have not restricted the format of $\mathcal{L}$ at all, except that each token must have a particular interpretation. However, any algorithm that works on this sort of model will be forced to iterate over the entire language set $\mathcal{L}$, which has the potential of being quite large. A more informative model needs to say something about what $\mathcal{L}$ looks like. Alternatively, it could also be effective if something could be learned or summarized about the set $\mathcal{L}$ once and then used.

D.2.1 Example: Abstract States

One natural way for a coach to communicate about the world is in terms of abstract states. This section will develop the model more fully for a language that can speak in terms of abstract states.

First, we'll define the set $\tilde{\mathcal{S}}$ as the set of abstract states. Two functions Abs and $Spec$ (abstraction and specialization) will define the relationship between $\tilde{\mathcal{S}}$ and $\mathcal{S}$.

$$Abs\colon\ \mathcal{S} \to \tilde{\mathcal{S}} \qquad \text{each state has an abstract state} \tag{D.7}$$

$$Spec\colon\ \tilde{\mathcal{S}} \to \mathcal{P}(\mathcal{S}) \qquad Spec(\tilde{s}) = \{s \in \mathcal{S} | Abs(s) = \tilde{s}\} \tag{D.8}$$

The parts of the model impacted are $\mathcal{L}$, L, $\mathcal{Y}$, and Y.

$\mathcal{L}$ In general, we could make no restriction on this set. It may be an interesting model to say that the coach can, for every abstract state define any other abstract state as a goal. This restriction would create a more structured language set.

L We will define a function $\tilde{L}$ which decodes the language into abstract states.

$$\tilde{L}\colon\ \mathcal{L} \to \left(\langle\tilde{\mathcal{S}}, \tilde{\mathcal{S}}\rangle \times \langle\tilde{\mathcal{S}}, \tilde{\mathcal{S}}\rangle\right) \tag{D.9}$$

The first tuple represents the advice, "If you are in the first (abstract) state, your goal is to get to the second abstract state." The second tuple is the removal of advice. This can be translated to L in the natural way.

$\mathcal{Y}$ The advice summary states can be simplified somewhat

$$\mathcal{Y} := \{y \mid y\colon\ \tilde{\mathcal{S}} \to \mathcal{P}(\tilde{\mathcal{S}})\} \tag{D.10}$$

Y This is defined in the natural way which we will not write out here.

D.2.2 Agent Ability

One of the challenges of the coaching problem is to learn about and adapt to the agents being coached. This section covers some preliminary thoughts about how this idea fits into this formal model. The ideas here are in general less developed than in the other sections.

Let's start by making some simplifying assumptions. Assume that the coach can and will only provide one goal state for each state of the world. Depending on the structure of the language, this may or may not be possible.

Second, we assume that the agent works on the given goal for a state until that goal is achieved. The problem of when goals are reevaluated is a very tricky one, and for the moment we will just ignore it.

A model of an agent can then be, for each state and given goal, the expected time to reach a goal state, the probability distribution over goal states, and the expected reward obtained along the way.

Formally, we can have agent model M as a function:

$$M\colon\ \mathcal{S} \times \mathcal{P}(\mathcal{S}) \to \mathbb{R} \times Pr(\mathcal{S}) \times \mathbb{R} \tag{D.11}$$

where Pr represents the set of probability distributions over its argument.

If the agent is not learning, this agent model has fixed values. The agent model and world dynamics can be used to establish the value of an advice summary state. If the coach can calculate the best advice summary state, it should be able to calculate a series of language utterances to transform the advice summary state.

D.3 Adding Multiple Agents

There are a variety of models to capture multi-agent systems. The model we present here follows the DEC-MDP [Bernstein et al., 2000]. Each agent will have a state and the global state of the world (which the coach will see) is determined by the conjunction of all the agents' states. We assume that the coach has broadcast communication with the agents and that the agents do not communicate among themselves.

We will present the model based on the general model presented in Section D.1. However, the specification in Section D.2 would apply equally well to this multi-agent model.

The model is a triple of tuples. First is for the agent, second is shared, and the third is for the coach: $\langle\langle\langle\mathcal{S}_1, \mathcal{A}_1\rangle, \ldots, \langle\mathcal{S}_n, \mathcal{A}_n\rangle\rangle, \langle\mathcal{L}, \Delta\rangle, \langle n, T, R, \mathcal{Y}, Y, y_0\rangle\rangle$

$\mathcal{S}_i$ Each agent i has a local state space. Define $\mathcal{S} := \cup\mathcal{S}_i$.

$\mathcal{A}_i$ Each agent i has a set of actions. Define the joint action space $\mathcal{A} := \cup\mathcal{A}_i$.

n The number of agents.

$T\colon\ \mathcal{S}\times\mathcal{A}\times\mathcal{S}\to\mathbb{R}$ The transition function. Given a joint state of the world and a joint action of all the agents, gives a probability distribution over next states.

$R\colon\ \mathcal{S}\times\mathcal{A}\to\mathbb{R}$ The reward function, which is determined by the joint state and joint action.

$\mathcal{Y}$ The set of advice summary states (same as for single agent model).

$Y\colon\ \mathcal{Y}\times\mathcal{L}\to\mathcal{Y}$ The language to advice summary interpretation function. Note that we have removed the dependence upon $\mathcal{S}$; the agents may not know the complete world state, so the advice interpretation can not depend upon it.

$y_0\in\mathcal{Y}$ The initial advice summary state (same as for single agent model).

$\mathcal{L}$ The language set, the set of all things that a coach could say (same as for single agent model).

Δ The interval at which the coach can talk.

We will use s_t^i to represent the local state of agent i at time t, and similarly with actions. Let $s_t = \cup s_t^i$.

We can now describe how the world evolves. The agent and the coach both know the current advice summary state $y_{t-1}\in\mathcal{Y}$. We are in a state of the world ($s_t = \cup s_t^i \in \mathcal{S}$). The coach uses y_{t-1} and s_t (joint state of all agents) to pick something to say ($l_t\in\mathcal{L}$). Note that this means the the coach's policy is $\pi_c\colon\ \mathcal{Y}\times\mathcal{S}\to\mathcal{L}$. The language to advice summary function is used to compute the new summary state $y_t = Y(y_{t-1}, l_t)$ (the coach and every agent can do this computations without explicitly sharing anything but l_t). Each agent i then uses y_t and s_t^i to pick an action from $\mathcal{A}_i$. Note that this means the agent's policy is $\pi_a^i\colon \mathcal{Y} \times \mathcal{S}_i \to \mathcal{A}_i$.

The agents' actions ($a_t = \cup a_t^i$) determines a state transition (s_{i+1}) and reward (r_t). Both the agents and coach then receive reward r_t.

To summarize, the history of the world here can be represented as a sequence of tuples $\langle s_t, l_t, y_t, a_t, r_t \rangle$ where

$s_t = \cup s_t^i$ The state at time t. The joint state s_t is determined by the agents' local states s_t^i.

l_t What the coach says at time t. $\emptyset$ can be used represent that the coach could not say anything.

y_t The advice summary state at time t (updated to include l_t)

$a_t = \cup a_t^i$ The agents' actions at time t

r_t The reward received at time t

A diagram describing how the history variables evolve over time in Figure D.2.

D.4 Future Extensions

While we believe these models are progress, there is much that could be added:

- Agents talking to the coach. All models assume unidirectional (explicit) communication.

- There are no time-delayed messages. This allows to coach to have an impact simply by telling the agent what to do right now or what the state of the world is (in the multi-agent case).

- Our more specialized model no longer allows the coach to talk about doing specific actions. How much of a restriction is this? One can certainly come up with domains where this is a problem (an agent has two actions which can go to the same states, but with different probabilities).

- The more specific model does not allow the coach to say things like "State 1 is better than state 2 which is better than state 3."

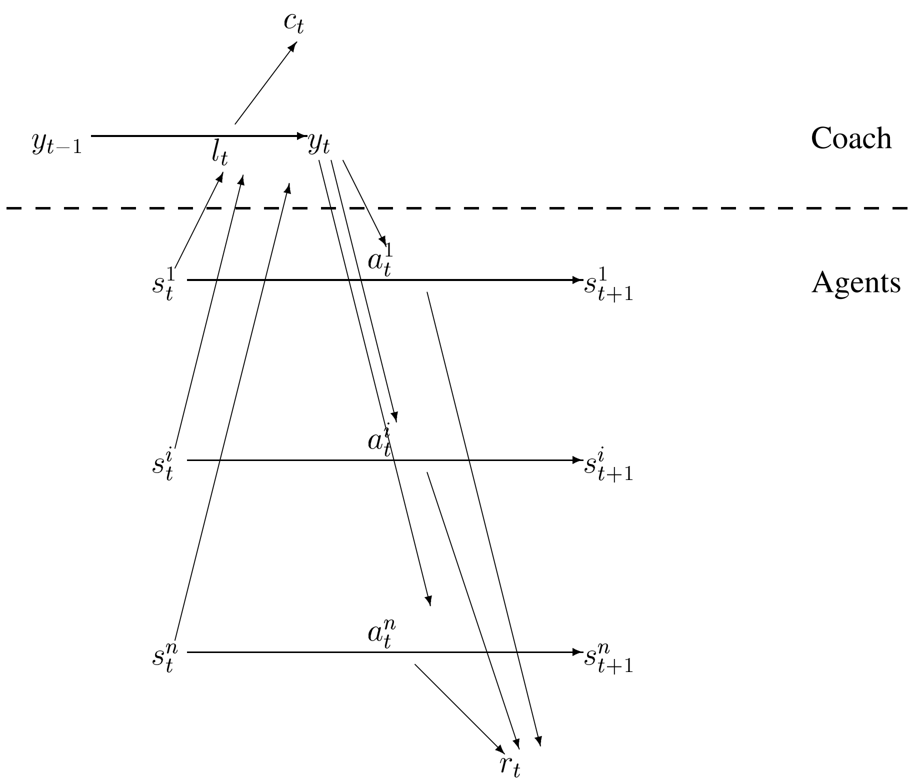


Figure D.2: Diagram describing history variable evolution over time in multi-agent model.

- It might be nice to characterize a language based on the structure provided. For example, you can characterize the "granularity" of a language (for either states or goals) in terms how small a set of states can be used. Also, an "options" function could say how many different goals the language provides you for each state.

  Characterizations such as these could be very useful for classifying languages and developing and/or analyzing algorithms.

Bibliography

This thesis collects and extends various previous publications by the author. In particular:

- Chapter 3 is based on Riley and Veloso [2003, 2004b].
- Chapter 4 is an extended version of Riley and Veloso [2004a], with more algorithmic detail and experiments.
- Chapter 5 collects and expands on several previous papers [Riley and Veloso, 2001a, 2002b,a, Riley et al., 2002b,a].
- Chapter 6 is based on Riley and Veloso [2001b, 2002a].

The number(s) at the end of an entry indicate what section(s) of the thesis use that citation.

Mazda Ahmadi, Abolfazl Keighobadi Lamjiri, Mayssam M. Nevisi, Jafar Habibi, and Kambiz Badie. Using a two-layered case-based reasoning for prediction in soccer coach. In *Proceedings of the International Conference on Machine Learning; Models, Technologies and Applications (MLMTA'03)*, pages 181–185, 2003. 7.4

John R. Anderson, C. Franklin Boyle, Albert T. Corbett, and Matthew W. Lewis. Cognitive modeling and intelligent tutoring. *Artificial Intelligence*, 42:7–49, 1990. 7.2

Elisabeth Andre, Kim Binsted, Kumiko Tanaka-Ishii, Sean Luke, Gerd Herzog, and Thomas Rist. Three robocup simulation league commentator systems. *AI Magazine*, pages 57–66, Spring 2000. 7.4

Christopher G. Atkeson and Stefan Schaal. Robot learning from demonstration. In Jr Douglas H. Fisher, editor, *Proceedings of the Fourteenth International Conference on Machine Learning (ICML-97)*, pages 12–20. Morgan Kaufmann, 1997. 7.1

Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The non-stochastic multi-armed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002. 5.1.2, 8.4.4

Paul Bakker and Yasuo Kuniyoshi. Robot see, robot do: An overview of robot imitation. In *AISB96 Workshop on Learning in Robots and Animals*, pages 3–11, Brighton,UK, 1996. 7.1

T. Balch and R. C. Arkin. Behavior-based formation control for multirobot teams. *IEEE Transactions on Robotics and Automation*, 14(6):926–939, 1998. 5.2, 5.2.1

Tucker Balch, Peter Stone, and Gerhard Kraetzschmar, editors. *RoboCup-2000: Robot Soccer World Cup IV*. Springer Verlag, Berlin, 2001. 5.2.3

Daniel S. Bernstein, Shlomo Zilberstein, and Neil Immerman. The complexity of decentralized control of markov decision processes. In *Proceedings of the Sixteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-2000)*, pages 32–37, 2000. 7.7, 8.4.5, D.3

Darse Billings, Denis Papp, Jonathan Schaeffer, and Duane Szafron. Opponent modeling in poker. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 493–498, Madison, WI, 1998. AAAI Press. 7.3

Daniel Borrajo and Manuela Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal, Special Issue on Lazy Learning*, 10:1–34, 1996. 7.6

Craig Boutilier. Planning, learning and coordination in multiagent decision processes. In *Theoretical Aspects of Rationality and Knowledge*, 1996. 8.4.5

Craig Boutilier. Sequential optimality and coordination in multiagent systems. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 478–485, 1999. 7.7, D

Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1):49–107, 2000. 7.5

Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. Context-specific independence in bayesian networks. In *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI-1996)*, pages 115–123, 1996. 7.5

Craig Boutilier and Martin L. Puterman. Process-oriented planning and average-reward optimality. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1995. 7.5

Michael Bowling, Brett Browning, and Manuela Veloso. Plays as effective multiagent plans enabling opponent-adaptive play selection. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, 2004. 7.7

David Carmel and Shaul Markovitch. Incorporating opponent models into adversary search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Portland, OR, 1996. 5.1.1, 7.3, 8.4.4

David Carmel and Shaul Markovitch. Model-based learning of interaction strategies in multiagent systems. *Journal of Experimental and Theoretical Artificial Intelligence*, 10 (3):309–332, 1998. 1.1, 5.1.1, 7.3

Paul Carpenter, Patrick Riley, Gal Kaminka, Manuela Veloso, Ignacio Thayer, and Robert Wang. ChaMeleons-01 team description. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup-2001: Robot Soccer World Cup V*, number 2377 in Lecture Notes in Artificial Intelligence, pages 503–506. Springer-Verlag, Berlin, 2002. 6.4.2

Paul Carpenter, Patrick Riley, Manuela Veloso, and Gal Kaminka. Integration of advice in an action-selection architecture. In Gal A. Kaminka, Pedro U. Lima, and Raul Rojas, editors, *RoboCup-2002: Robot Soccer World Cup VI*, number 2752 in Lecture Notes in Artificial Intelligence, pages 195–205. Springer Verlag, Berlin, 2003. 1.2

P. Cheeseman, J. Kelly, M. Self, J. Stutz, W. Taylor, and D. Freeman. Autoclass: A bayesian classification system. In *Proceedings of the Fifth International Conference on Machine Learning (ICML-88)*, pages 54–64, San Francisco, June 1988. Morgan Kaufmann. 5.2.2

Mao Chen, Klaus Dorer, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Jan Murray, Itsuki Noda, Oliver Obst, Pat Riley, Timo Steffens, Yi Wang, and Xiang Yin. *Soccer Server Manual*. RoboCup Federation, `http://sserver.sourceforge.net/`, 2001. 2.2, 2.2.2, A.3

David Maxwell Chickering, David Heckerman, and Christopher Meek. A beyesian approach to learning bayesian networks with local structure. Technical Report MSR-TR-97-07, Microsoft Research, 1997. 7.5

Jeffrey Clouse. Learning from an automated training agent. In Diana Gordon, editor, *Working Notes of the ICML '95 Workshop on Agents that Learn from Other Agents*, Tahoe City, CA, 1995. 7.1

Jeffrey A. Clouse and Paul E. Utgoff. A teaching method for reinforcement learning. In *Proceedings of the Ninth International Conference on Machine Learning (ICML-92)*, pages 92–101, 1992. 7.1

Coach Mailing List. Robocup coach mailing list. `http://robocup.biglist.com/coach-l/`,`http://mailman.cc.gatech.edu/mailman/listinfo/robocup-coach-l`. 2.2.2, A

Andreas Birk Silvia Coradeschi and Satoshi Tadokoro, editors. *RoboCup-2001: Robot Soccer World Cup V*. Springer Verlag, Berlin, 2002. 5.2.3, 6.4.2

A. Corbett and J. Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction*, 4:253–278, 1995. 7.2

T. Cover and J. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. John Wiley & Sons, New York, 1991. 5.1.4

DARPA. Software for distributed robotics (SDR) indoor reconnaissance experiment/demonstration. `http://www.darpa.mil/ipto/solicitations/closed/02-14_PIP.htm`, 2002. 1.4, 1.4.1

Kerstin Dautenhahn. Getting to know each other—artificial social intelligence for autonomous robots. *Robotics and Autonomous Systems*, 16:333–356, 1995. 7.1

T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 76(1–2):3–74, 1989. 7.5

Richard Dearden and Craig Boutilier. Abstraction and approximate decision theoretic planning. *Artificial Intelligence*, 89(1):219–283, 1997. 1.1

Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991. 6.1, 6.2.1, 6.2.2

Jörg Denzinger and Jasmine Hamdan. Improving modeling of other agents using stereotypes and compactification of observations. In *Proceedings of the Third Autonomous Agents and Multi-Agent Systems Conference*, 2004. 7.3

Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000. 7.5

R.J. Doyle, D.J. Atkinson, and R.S. Doshi. Generating perception requests and expectations to verify the executions of plans. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, 1986. 8.4.3

Kurt Driessens and Sašo Džeroski. Integrating guidance into relational reinforcement learning. *Machine Learning*, 57:271–304, 2004. 7.1

Christian Drücker, Sebastian Hübner, Esko Schmidt, Ubbo Visser, and Hans-Georg Weland. Virtual werder. In P. Stone, T. Balch, and G. Kraetzschmar, editors, *RoboCup 2000: Robot Soccer. World Cup IV*, volume 2019 of *Lecture Notes in Artificial Intelligence*, pages 421–424. Springer-Verlag, 2001. 7.4

Tara A. Estlin and Raymond J. Mooney. Learning to improve both efficiency and quality of planning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1127–1232, 1997. 7.6

Lance Fortnow and Duke Whang. Optimality and domination in repeated games with bounded players. In *Proceedings of the 26th ACM Symposium on the Theory of Computing*, pages 741–749, New York, 1994. ACM. 7.3

Yoav Freund, Michael Kearns, Yishai Mansour, Dana Ron, and Ronitt Rubinfeld. Efficient algorithms for learning to play repeated games against computationally bounded adversaries. In *Proc. 36th IEEE Conference on Foundations of Computer Science*, 1995. 7.3

Yoav Freund and Robert E. Schapire. Adaptive game playing using multiplicative weights. *Games and Economic Behavior*, 29:79–103, 1999. 7.3

Dan Geiger and David Heckerman. Knowledge representation and inference in similarity networks and Bayesian multinets. *Artificial Intelligence*, 82:45–74, 1996. 7.5

Piotr J. Gmytrasiewicz and Edmund H. Durfee. A rigorous, operational formalization of recursive modeling. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, 1995. 5.1, 7.3

Sally A. Goldman and Michael J. Kearns. On the complexity of teaching. *Journal of Computer and System Sciences*, 50(1):20–31, 1995. 7.2

Sally A. Goldman and H. David Mathias. Teaching a smarter learner. *Journal of Computer and System Sciences*, 52(2):255–267, 1996. 7.2

Sally A. Goldman, R. L. Rivest, and R. E. Schapire. Learning binary relations and total orders. *SIAM Journal on Computing*, 22(5):1006–1034, 1993. 7.2

Barbara Grosz and Sarit Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357, 1996. 7.7

Carlos Guestrin and Geoffrey Gordon. Distributed planning in hierarchical factored mdps. In *Proceedings of the Eighteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-2002)*, 2002. 7.5

Carlos Guestrin, Daphne Koller, and Ronald Parr. Multiagent planning with factored mdps. In *Advances in Neural Information Processing Systems 14*, 2001. 7.5

Jafar Habibi, Ehsan Chiniforooshan, A. HeydarNoori, M. Mirzazadeh, MohammadAli Safari, and HamidReza Younesi. Coaching a soccer simulation team in robocup environment. In *Proceedings of the First EurAsian Conference on Information and Communication Technology*, pages 117–126. Springer-Verlag, 2002. ISBN 3-540-00028-3. 7.4

Kwun Han and Manuela Veloso. Automated robot behavior recognition applied to robotic soccer. In *Proceedings of IJCAI-99 Workshop on Team Behaviors and Plan Recognition*, 1999. 7.3

Scott B. Huffman and John E. Laird. Flexibly instructable agents. *Journal of Artificial Intelligence Research*, 3:271–324, November 1995. 8.4.2

S.S. Intille and A.F. Bobick. A framework for recognizing multi-agent action from visual evidence. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 518–525. AAAI Press, 1999. 7.7

Jeffrey Jackson and Andrew Tomkins. A computational model of teaching. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 319–326. ACM Press, July 1992. 7.2

N. Jennings. Controlling cooperation problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75, 1995. 7.7

Rune M. Jensen, Manuela M. Veloso, and Randy E. Bryant. Fault Tolerant Planning: Toward Probabilistic Uncertainty Models in Symbolic Non-Deterministic Planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, 2004. 4.2.2

M. Kaiser, H. Friedrich, and R. Dillmann. Obtaining good performance from a bad teacher. In *ICML Workshop on Programming by Demonstration*, Tahoe City, California, 1995. 7.1

Gal Kaminka, Mehmet Fidanboylu, Allen Chang, and Manuela Veloso. Learning the sequential behavior of teams from observations. In *RoboCup 2002: Robot Soccer World Cup VI*, volume 2752 of *Lecture Notes in Artificial Intelligence*, pages 111–125. Springer-Verlag, 2003. 7.3

Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The RoboCup synthetic agent challenge. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 24–49, San Francisco, CA, 1997. 2.2

Daphne Koller and Brian Milch. Multi-agent influence diagrams for representing and solving games. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001)*, pages 1027–1034, 2001. 7.5

Daphne Koller and Ronald Parr. Computing factored value functions for policies in structured MDPs. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999. 7.5

K. Konolige, D. Fox, C. Ortiz, A. Agno, M. Eriksen, B. Limketkai, J. Ko, B. Morisset, D. Schulz, B. Stewart, and R. Vincent. Centibots: Very large scale distributed robotic teams. In *Proceedings of the 9th International Symposium on Experimental Robotics 2004 (ISER '04)*, 2004. 1.4.1

Rich Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1): 35–77, 1985. 7.5

Gregory Kuhlmann, Peter Stone, and Justin Lallinger. The champion UT Austin Villa 2003 simulator online coach team. In Daniel Polani, Brett Browning, Andrea Bonarini, and Kazuo Yoshida, editors, *RoboCup-2003: Robot Soccer World Cup VII*. Springer Verlag, Berlin, 2004. (to appear). 1, 7.4

Y. Kuniyoshi., M. Inaba, and H. Inoue. Learning by watching: Extracting reusable task knowledge from visual observation of human performance. *IEEE. Trans. on Robotics and Automation*, 10(6):799–822, 1994. 7.1

John Laird, Paul S Rosenbloom, and Allen Newell. *Universal subgoaling and chunking : the automatic generation and learning of goal hierarchies*. Kluwer Academic Publishers, Boston, 1986a. 7.6

John E. Laird. It knows what you're going to do: Adding anticipation to a quakebot. In *Proceedings of the Fifth International Conference on Autonomous Agents (Agents-2001)*, 2001. 7.3

John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: an architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987. 7.6

John E. Laird, Paul S Rosenbloom, and Allen Newell. Towards chunking as a general learning mechanism. In *Proceedings of the Fourth National Conference on Artificial Intelligence (AAAI-84)*, pages 188–192, 1984. 7.6

John E. Laird, Paul S Rosenbloom, and Allen Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986b. 1.1, 7.6

Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991. 6.3.1

Long-Ji Lin. Self-improving reactive agent based on reinforcement learning, planning, and teaching. *Machine Learning*, 8:293–321, 1992. 7.1

Richard Maclin and Jude W. Shavlik. Creating advice-taking reinforcement learners. *Machine Learning*, 22:251–282, 1996. 1.1

H. David Mathias. A model of interactive teaching. *Journal of Computer and System Sciences*, 54(3):487–501, 1997. URL `citeseer.nj.nec.com/mathias97model.html`. 7.2

David McAllester and Peter Stone. Keeping the ball from CMUnited-99. In Peter Stone, Tucker Balch, and Gerhard Kraetszchmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*, Berlin, 2001. Springer Verlag. 6.2.3, 6.3.1

Andrea Miene, Ubbo Visser, and Otthein Herzog. Recognition and prediction of motion situations based on a qualitative motion description. In Daniel Polani, Andrea Bonarini, Brett Browning, and Kazuo Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Artificial Intelligence*, pages 77–88. Springer-Verlag, 2004. 7.3

Michael S. Miller, Jianwen Yin, Richard A. Volz, Thomas R. Ioerger, and John Yen. Training teams with collaborative agents. In *Fifth International Conference on Intelligent Tutoring Systems (ITS'2000)*, pages 63–72, 2000. 7.2

Steve Minton. *Learning Search Control Knowledge: An Explanation-Based Approach*. Kluwer Academic, 1988. 1.1, 3.4, 4.2.3, 7.6

Tom Mitchell. Learning and problem solving. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83)*, pages 1139–1151. Morgan Kaufmann, 1983. 7.6

Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986. 7.6

Takeshi Morimoto. How to develop a robocuprescue agent. `http://ne.cs.uec.ac.jp/~morimoto/rescue/manual/index.html`, Nov 2002. Ed. 1.00. 1.4.2

Paul Morris and Nicola Muscettola. Execution of temporal plans with uncertainty. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 491–496. AAAI Press/The MIT Press, 2000. 6.2.1, 7.7

Nicola Muscettola, Paul Morris, and Ioannis Tsamardinos. Reformulating temporal plans for efficient execution. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, 1998. 6.2.3, 6.2.3, 7.7

A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML-99)*, 1999. 7.1

Itsuki Noda, Hitoshi Matsubara, Kazuo Hiraki, and Ian Frank. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12(2–3):233–250, 1998. 2.2

Ron Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10*, Cambridge, MA, 1998. MIT Press. 7.5

Leonid Peshkin, Kee-Eung Kim, Nicolas Meuleau, and Leslie Pack Kaelbling. Learning to cooperate via policy search. In *Proceedings of the Sixteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-2000)*, 2000. 7.7, 8.4.5

Martha C. Polson and J. Jeffrey Richardson, editors. *Foundations of Intelligent Tutoring Systems*. Lawrence Erlbaum Associates, 1988. 1.1, 7.2

Bob Price and Craig Boutilier. Implicit imitation in multiagent reinforcement learning. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML-99)*, pages 325–334. Morgan Kaufmann, San Francisco, CA, 1999. 7.1

Bob Price and Craig Boutilier. Imitation and reinforcement learning in agents with heterogeneous actions. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000)*, 2000. 7.1

Martin L. Puterman. *Markov Decision Processes*. John Wiley & Sons, New York, 1994. 1.1, 2.1, 4.2.3, 7.5

David Pynadath and Milind Tambe. The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, 16:389–423, 2002a. 7.7, 8.4.5, D

David Pynadath and Milind Tambe. Multiagent teamwork: Analyzing the optimality and complexity of key theories and models. In *Proceedings of the First Autonomous Agents and Multi-Agent Systems Conference*, 2002b. 8.4.5

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993. 5.2.2

Taylor Raines, Milind Tambe, and Stacy Marsella. Automated assistant to aid humans in understanding team behaviors. In *Proceedings of the Fourth International Conference on Autonomous Agents (Agents-2000)*, 2000. 7.3, 7.4

Luis Paulo Reis and Nuno Lau. COACH UNILANG - a standard language for coaching a (robo)soccer team. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup-2001: Robot Soccer World Cup V*, volume 2377 of *Lecture Notes in Artificial Intelligence*, pages 183–192. Springer-Verlag, 2002. 7.4

J. Rickel and W.L. Johnson. Animated agents for procedural training in virtual reality: Perception, cognition, and motor control. *Applied Artificial Intelligence*, 13:343–382, 1999. 7.2

J. Rickel and W.L. Johnson. Extending virtual humans to support team training in virtual reality. In G. Lakemeyer and B. Nebel, editors, *Exploring Artificial Intelligence in the New Millennium*. Morgan Kaufmann, San Francisco, 2002. 7.2

Patrick Riley. MPADES: Middleware for parallel agent discrete event simulation. In Gal A. Kaminka, Pedro U. Lima, and Raul Rojas, editors, *RoboCup-2002: Robot Soccer World Cup VI*, number 2752 in Lecture Notes in Artificial Intelligence, pages 162–178. Springer Verlag, Berlin, 2003. *RoboCup Engineering Award*. 2.1

Patrick Riley and George Riley. SPADES — a distributed agent simulation environment with software-in-the-loop execution. In S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, editors, *Winter Simulation Conference Proceedings*, volume 1, pages 817–825, 2003. 2.1

Patrick Riley, Peter Stone, David McAllester, and Manuela Veloso. ATT-CMUnited-2000: Third place finisher in the robocup-2000 simulator league. In P. Stone, T. Balch, and G. Kreatzschmarr, editors, *RoboCup-2000: Robot Soccer World Cup IV*, number 2019 in Lecture Notes in Artificial Intelligence, pages 489–492. Springer, Berlin, 2001. 6.4.2

Patrick Riley and Manuela Veloso. Coaching a simulated soccer team by opponent model recognition. In *Proceedings of the Fifth International Conference on Autonomous Agents (Agents-2001)*, pages 155–156, 2001a. D.4

Patrick Riley and Manuela Veloso. Planning for distributed execution through use of probabilistic opponent models. In *IJCAI-2001 Workshop PRO-2: Planning under Uncertainty and Incomplete Information*, pages 18–26, 2001b. D.4

Patrick Riley and Manuela Veloso. Planning for distributed execution through use of probabilistic opponent models. In *Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS-2002)*, pages 72–81, 2002a. *Best Paper Award*. D.4

Patrick Riley and Manuela Veloso. Recognizing probabilistic opponent movement models. In A. Birk, S. Coradeschi, and S. Tadokoro, editors, *RoboCup-2001: Robot Soccer World Cup V*, number 2377 in Lecture Notes in Artificial Intelligence, pages 453–458. Springer Verlag, Berlin, 2002b. (extended abstract). D.4

Patrick Riley and Manuela Veloso. An overview of coaching with limitations. In *Proceedings of the Second Autonomous Agents and Multi-Agent Systems Conference*, pages 1110–1111, 2003. D.4

Patrick Riley and Manuela Veloso. Advice generation from observed execution: Abstract Markov decision process learning. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-2004)*, 2004a. D.4

Patrick Riley and Manuela Veloso. Coaching advice and adaptation. In Daniel Polani, Andrea Bonarini, Brett Browning, and Kazuo Yoshida, editors, *RoboCup-2003: The Sixth RoboCup Competitions and Conferences*. Springer Verlag, Berlin, 2004b. (to appear). D.4

Patrick Riley, Manuela Veloso, and Gal Kaminka. An empirical study of coaching. In H. Asama, T. Arai, T. Fukuda, and T. Hasegawa, editors, *Distributed Autonomous Robotic Systems 5*, pages 215–224. Springer-Verlag, 2002a. D.4

Patrick Riley, Manuela Veloso, and Gal Kaminka. Towards any-team coaching in adversarial domains. In *Proceedings of the First Autonomous Agents and Multi-Agent Systems Conference*, pages 1145–1146, 2002b. D.4

RoboCup. Robocup web page. `http://www.robocup.org`. 2.2

RoboCup Rescue. Robocup rescue. `http://www.rescuesystem.org/robocuprescue/`. 1.4, 1.4.2

C. Sammut, S. Hurst, D. Kedzier, and D. Michie. Learning to fly. In *Proceedings of the Ninth International Conference on Machine Learning (ICML-92)*. Morgan Kaufmann, 1992. 7.1

Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53 (2–3):273–290, 1992. 7.3

Guy Shani, Ronen I. Brafman, and David Heckerman. An MDP-based recommender system. In *Proceedings of the Eighteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-2002)*, pages 453–460, 2002. 7.5

Ayumi Shinohara and Satoru Miyano. Teachability in computational learning. *New Generation Computing*, 8:337–437, 1991. 7.2

Timo Steffens. Feature-based declarative opponent-modelling in multi-agent systems. Master's thesis, Institute of Cognitive Science Osnabrück, 2002. URL `citeseer.nj.nec.com/steffens02featurebased.html`. 7.4

A. Stentz. Optimal and efficient path planning for partially known environments. In *Proceedings of IEEE International Conference on Robotics and Automation*, May 1994. 6.3.1

Peter Stone. *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. Intelligent Robotics and Autonomous Agents. MIT Press, 2000. 1.1, 6.2.3

Peter Stone, Patrick Riley, and Manuela Veloso. The CMUnited-99 champion simulator team. In Veloso, Pagello, and Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*, number 1856 in Lecture Notes in Artificial Intelligence, pages 35–48. Springer, Berlin, 2000. 5.1, 5.2.1, 5.2.3, 6.1, 6.2.3, 6.4.2

Peter Stone and Manuela Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 110 (2):241–273, June 1999. 5.2.1, 6.3.2

Peter Stone, Manuela Veloso, and Patrick Riley. The CMUnited-98 champion simulator team. In Asada and Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*, number 1604 in Lecture Notes in Artificial Intelligence, pages 61–75. Springer, 1999. 5.1, 6.1

Dorian Šuc and Ivan Bratko. Skill reconstruction as induction of LQ controllers with subgoals. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 914–919, 1997. 7.1

R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999. 4.4.3, 7.5

Tomoichi Takahashi. Kasugabito III. In Veloso, Pagello, and Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*, number 1856 in Lecture Notes in Artificial Intelligence, pages 592–595. Springer-Verlag, Berlin, 2000. 7.4

Yasutake Takahashi, Koichi Hikita, and Minoru Asada. A hierarchical multi-module learning system based on self-interpretation of instructions by coach. In Daniel Polani, Brett Browning, A. Bonarini, and K. Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Artificial Intelligence*, pages 576–583, 2004. 7.1

M. Tambe and P.S. Rosenbloom. Resc: An approach for dynamic, real-time agent tracking. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1995. 7.3

Milind Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7: 83–124, 1997. 1.1, 7.7

Kumiko Tanaka-Ishii, Itsuki Noda, and Ian Frank et.al. Mike: An automatic commentary system for soccer — system design and control —. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS-98)*, pages 285–292, Paris, France, 1998. 7.4

Manuela Veloso, Michael Bowling, and Peter Stone. Anticipation as a key for collaboration in a team of agents: A case study in robotic soccer. In *Proceedings of SPIE Sensor Fusion and Decentralized Control in Robotic Systems II*, volume 3839, Boston, September 1999. 5.1

Manuela Veloso, Peter Stone, Kwun Han, and Sorin Achim. CMUnited: A team of robotic soccer agents collaborating in an adversarial environment. In Hiroaki Kitano, editor, *RoboCup-97: The First Robot World Cup Soccer Games and Conferences*, pages 242–256. Springer Verlag, Berlin, 1998. 5.1, 6.1

Manuela M. Veloso, Jaime Carbonell, M. Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The prodigy architecture,. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995. 7.6

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3): 279–292, 1992. 3.1.1

Gerhard Weiss, editor. *Multiagent systems : a modern approach to distributed artificial intelligence*. MIT Press, Cambridge, Mass., 1999. 6.3.2

Eric Wiewiora, Garrison Cottrell, and Charles Elkan. Principled methods for advising reinforcement learning agents. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2001)*, 2003. 7.1

Michael Wünstel, Daniel Polani, Thomas Uthmann, and Jürgen Perl. Behavior classification with self-organizing maps. In *RoboCup-2000*, volume 2019 of *Lecture Notes in Artificial Intelligence*, pages 108–118. Springer Verlag, 2001. 7.3

Ping Xuan, Victor Lesser, and Shlomo Zilberstein. Communication decisions in multiagent markov decision processes: Model and experiments. In *Proceedings of the Fifth International Conference on Autonomous Agents (Agents-2001)*, pages 616–623, 2001. 5.1.1, 7.7, 8.4.5