Thresholded-Rewards Decision Problems: Acting Effectively in Timed Domains

Colin McMillen

CMU-CS-09-112

April 2, 2009

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

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

This research was sponsored by Rockwell Scientific under grant number B4U528968, Department of the Interior under grant number NBCH1040007, L3 Communication Integrated Systems, L.D., under grant number 4500244745, SRI International under grant number 55-000691 and 03-000211, DARPA, Intelligent Automations, Inc., under grant number 6541, Lockheed Martin Corporation under grant number 8100001629, and U.S. Army Research Office under grant number DAAD-190210389. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Markov Decision Processes, Thresholded Rewards, Limited Time, Zero-Sum Games, Robotics, Multi-Robot Systems, Multi-Agent Systems, Robot Soccer, Capture the Flag, reCAPTCHA

Abstract

In timed, zero-sum games, *winning* against the opponent is more important than the final score. A team that is losing near the end of the game may choose to play aggressively to try to even the score before time runs out. In this thesis, we consider the problem of finding *optimal policies* in stochastic domains with limited time, some notion of score, and in complex environments, such as domains including opponents. This problem is relevant to many intelligent decision making tasks, not just games, as nearly every decision made in the real world depends on time. The work presented in this thesis has broad applications to domains possessing the key features of control under uncertainty, limited time, and some notion of score.

We introduce the concept of *thresholded-rewards problems* as a means to effectively reason about acting in domains with limited time and with some notion of score, progress, or intermediate reward. In a thresholded-rewards problem, the amount of *true reward* received is determined at the end of the time horizon, by applying an arbitrary threshold function to the amount of *intermediate reward* (e.g., score) accumulated during execution. We utilize Markov decision processes (MDPs) and semi-Markov decision processes (SMDPs) to model domains with stochastic actions. We introduce algorithms for finding optimal policies in MDPs and SMDPs with thresholded rewards. We also introduce heuristics for finding approximately optimal policies for thresholded-rewards MDPs. We analyze how a team should change strategy in response to an opponent whose behavior is initially unknown but slowly reveals itself during execution. We also introduce a sampling-based control algorithm that allows for effective action in domains in which rewards are hidden from the agent.

We perform controlled experiments to evaluate our algorithms in three timed domains. *Robot soccer* and *Capture the Flag* are timed, adversarial games in which two teams compete to be ahead in score at the end of the game. We further extend our approach to address the *reCAPTCHA* domain, in which we are given a set of words that need to be transcribed before some time deadline. The control problem consists of maximizing the probability that all the words have been transcribed before the deadline. Through our theoretical and experimental results, we show that the algorithms presented in this thesis enable effective action in stochastic domains with limited time and some notion of score.

Acknowledgements

I would like to thank my advisor, Manuela Veloso, for six years of guidance and support. I am also grateful to my committee members, Stephen Smith, Drew Bagnell, and Michael Littman, for their advice and suggestions.

I would also like to thank the many members of the robot soccer lab over the years, especially Sonia Chernova, Doug Vail, Jim Bruce, and Paul Rybski. Your support as friends and collaborators has kept me (mostly) sane through countless deadlines and competitions.

I am also indebted to all the teachers—from kindergarten through graduate school—who have inspired me to learn for the last 23 years. In particular, I would like to thank Paul Hetchler and Bob O'Hara, my high-school physics and biology teachers, who inspired me to go into science; and Nikolaos Papanikolopoulos and Maria Gini, who got me started with robotics research as an undergraduate.

I am thankful to have the love and support of my parents, Shaun and Roseann, and my sister, Katie.

Finally, I am deeply indebted to my wife, Kristen, whose love, support, patience, and companionship have helped me throughout the entire Ph.D. process—but especially in the last few months, for encouraging me to push through to the end.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In timed, zero-sum games (such as soccer, basketball, American football, ice hockey, and most other team sports), winning against the opponent is what matters, regardless of the magnitude of the final score. Intuitively, it seems that a team which is losing near the end of the game should play aggressively, trying to even the score before time runs out. Such a team can be said to be *risk-taking*: willing to trade expected reward for a higher probability of winning (or at least tying) the game. Conversely, a team which is winning near the end of the game may want to play a *risk-averse* strategy, playing defensively to prevent the opponent from scoring. These sorts of strategies, which depend on score and time remaining, have been used widely in human sports.

I have worked on robot soccer since the fall of 2003, as part of the CMPack and CMDash entries to the RoboCup four-legged league. My particular focus has always been multi-robot communication, collaboration, and teamwork. Early on, I found that the strategies chosen by RoboCup teams were lacking one of the fundamental components commonly adopted by human strategists: reasoning about score and time. A robot playing in the first minute of a soccer match would behave exactly the same as a robot playing in the final minute of a soccer match.

In the RoboCup 2005 international competition, we introduced the first AIBO robot soccer team that autonomously changed strategy depending on the score of the game and the time remaining. We developed a distributed, play-based approach which allows our team to dynamically allocate team members to more defensive or more offensive roles as the game progresses [37–39]. With this approach, we used hand-coded applicability conditions to cause the team to act more aggressively or more defensively depending on the score and time. For example, for an aggressive play that assigns many robots to attacking roles, we might have an applicability condition like "This play is applicable when our team is losing and there is less than one minute remaining in the game." The main motivation for this thesis was the desire to eliminate these hand-coded rules and instead find an algorithm that derives policies for *optimal play selection* in timed, zero-sum domains such as robot soccer. The key insight is that teams in timed, zero-sum games need to maximize the probability of *winning*: being ahead in score when time runs out.

From this initial motivation, we came to understand that this problem is relevant to *many* intelligent decision making tasks, not just games: nearly every decision made daily by people in the real world depends on time. These time-dependent decisions are also rooted in uncertainty. However, people generally have some idea of the progress they are making toward their long-term goals. The work presented in this thesis therefore has broad applications beyond robot soccer; there are many real-world domains possessing the key features of uncertainty, limited time, and some notion of score.

Some related domains (not explicitly addressed in this thesis) include *election campaigning* and *retirement planning*. In election campaigning, political adversaries compete to be "ahead" (supported by a majority or plurality of voters) when time runs out (on election day). During the campaign season, the candidates can see their current "score" (e.g., as measured by opinion polls) and take actions to improve their score, including advocating certain policies and giving speeches in specific locations. In election campaigns, we see that candidates' strategies change significantly depending on their current score and time remaining. In retirement planning, people often have goals such as "I would like to maximize my probability of having k dollars saved by the time I am h years old." Here we again have a notion of "score" (current number of dollars saved) and a time deadline. In retirement planning, we see that investors' strategies change significantly based on the proximity of the time horizon. Typically, young investors invest the majority of their savings in stocks with a high expected value, while older investors will tend to invest more in bonds, which have a lower expected value than stocks but also lower variance, and therefore offer more predictable returns over a short time horizon.

The principal question addressed in this thesis is:

> **In stochastic domains with limited time, some notion of score, and possibly in the presence of adversaries, how can agents or teams act optimally, so as to maximize the probability of achieving their goals before the time deadline expires?**

Researchers in the operations research and game theory communities have previously addressed the problem of acting to maximize the probability of winning in time-limited, zero-sum games (e.g., [26, 59]). However, reasoning about time and score has not generally been addressed in rich real-world domains, such as robotic domains. In rich domains, it is easier to maximize score instead of maximizing the probability of winning: both from a computational complexity standpoint, and from the standpoint of implementation complexity (needing to write additional code). Through the work presented in this thesis, we aim to understand what can be gained by reasoning about score and time in such domains. We argue that there are significant gains that can be achieved by maximizing the probability of winning rather than maximizing expected score, and that these gains can be achieved with a tractable computational cost. Furthermore, we aim to reduce the implementation complexity of reasoning about score and time, by providing algorithms which automatically convert a score-maximizing model into an expanded model that includes score and time and explicitly maximizes the probability of winning.

In this thesis, we evaluate our algorithms in three timed domains. *Robot soccer* and *Capture the Flag* (CTF) are timed, adversarial games in which two teams compete to be ahead in score at the end of the game. In the *reCAPTCHA* domain, there is no adversary—we are given a set of w words that need to be transcribed before some time deadline, and need to maximize the probability that all w words have been transcribed before the deadline. These domains are described fully in Chapter 2. Below, we discuss the general domain features that are illustrated by our choice of these three domains.

1.1 Approach

The thesis question implicitly poses a control problem: given a stochastic domain with a fixed, finite time horizon, and some notion of score, we aim to find the optimal policy that maximizes the probability of achieving a specified amount of score before the time deadline expires. We assume that the useful state features for deriving the optimal policy are at a high level, such as the score of the game, the time remaining, and other domain features that have been significantly abstracted. We assume that these high-level features are known with near-certainty and that the state of the world is effectively fully observable.

In stochastic domains with full observability, Markov decision processes (MDPs) are a powerful tool for computing optimal policies. Therefore, MDPs are the basic building block on which we build most of the work presented in this thesis. Given a *base MDP* describing the stationary dynamics underlying a domain, we provide an algorithm that transforms this MDP into a *thresholded-rewards MDP* (TRMDP): an expanded MDP that explicitly represents score and time. In this expanded MDP, the amount of *true reward* received is determined by applying an arbitrary threshold function to the amount of *intermediate re-*

ward (i.e., score) accumulated during execution. We present an efficient algorithm that finds the optimal policy for this expanded MDP. Running this algorithm produces the optimal policy for the associated thresholded-rewards problem. In general, the resulting policy is non-stationary: the policy depends on the underlying state of the system as well as the current score and the amount of time remaining.

With MDPs, each action takes a fixed amount of time to execute. In this thesis, we also consider domains in which actions are temporally extended, requiring multiple time steps to complete. We therefore augment our models with semi-Markov actions, thereby introducing thresholded-rewards semi-MDPs (TRSMDPs). In further work, we consider the ramifications of playing against an opponent whose behavior is unknown to us *a priori*, and of acting in a domain in which the rewards are unknown at runtime. In each case, we introduce further extensions to our TRMDP algorithms which enable solutions to these additional challenges.

1.2 Domains

There are several general features shared by all three of the specific domains used in our empirical evaluation. The work presented in this thesis applies to domains possessing the following characteristics:

- **Finite time.** This thesis addresses domains in which there is a finite amount of time available. In domains with finite time, goals must be achieved before the time deadline expires.

- **Score.** Scores are intermediate rewards that represent progress made toward the final goal. In games like robot soccer and Capture the Flag, the scores are obvious: we get $+1$ reward every time we score and -1 reward every time the opponent scores. In the reCAPTCHA domain, we get one point of reward every time a word is successfully transcribed.

- **Thresholded rewards.** In domains with thresholded rewards, true reward is received only when the time horizon expires. When the time horizon expires, the amount of reward received is determined by applying an arbitrary threshold function to the amount of score (intermediate reward) received during execution.

- **Stochastic actions.** The effects of actions are uncertain; our algorithms therefore need to account for stochastic actions.

- **Full observability.** We assume that the useful state features for deriving the optimal policy are at a high level, such as the score of the game, the time remaining, and other domain features that have been significantly abstracted. We assume that these high-level features are known with near-certainty and that the state of the world is effectively fully observable. Our previous experience with RoboCup indicates that this assumption is justified.

In addition to the common features described above, each of the domains we study have additional features which create further challenges. Robot soccer and CTF are **adversarial** domains in which there are opponents that also manipulate the environment. In such domains, the optimal policy needs to take the behavior of the opponent into account. We initially assume that the behavior of the opponents is fully observable and known *a priori*; however, in Chapter 6, we relax this assumption, assuming instead that we have some means of recognizing the opponent's behavior online. Robot soccer and CTF are also have **teams** of individual agents working together. In this thesis, we treat a team as a single unit. Each action taken by a team corresponds to choosing a *play*: a team strategy that assigns *roles* to each team member. A *role* is a top-level behavior for a single agent.

Even though there are no opponents in the reCAPTCHA domain, reCAPTCHA still possesses the key features of score and time, and so the algorithms introduced in this thesis still apply to the reCAPTCHA domain. The reCAPTCHA domain has the additional challenge of **unknown rewards**: we are unable to observe the rewards received at execution time. Since the optimal policy for thresholded-rewards domains generally depends on the score, unknown rewards pose a significant challenge.

We present extensive descriptions of the robot soccer, CTF, and reCAPTCHA domains in Chapter 2.

1.3 Contributions

Concretely, the major contributions of this thesis are the following:

- The formal definition of *thresholded-rewards problems* as a means to analyze the tradeoffs between maximizing score and maximizing the true objective function (e.g., the probability of winning), in domains with limited time and some notion of score, progress, or intermediate reward.

- An algorithm which takes in a threshold function and a base MDP describing the stationary dynamics of a domain, and finds the optimal policy which maximizes the expected value of the given threshold function.
- Multiple heuristic approximation techniques for finding approximately optimal policies for thresholded-rewards MDPs.
- An exact algorithm for solving thresholded-rewards SMDPs, which accurately model domains in which actions are temporally extended or in which the amount of time taken to achieve a reward is drawn from an arbitrary distribution.
- The introduction of *incidental behavior recognition* as an interesting problem arising in domains with limited time and in which observations of the environment are received incidentally while a team is engaged in some other primary task.
- An analysis of how a team should change strategy in response to an opponent whose behavior is initially unknown but slowly reveals itself during execution.
- A sampling-based control algorithm that allows for effective action in domains in which rewards are hidden from the agent.
- Application and evaluation of these techniques to three different timed, finite-horizon domains, including experiments performed with our real robot soccer team.

1.4 Reader's Guide to the Thesis

The thesis is organized as follows:

- **Chapter 2** summarizes the three experimental domains that are used throughout the thesis: robot soccer, Capture the Flag, and reCAPTCHA.
- **Chapter 3** introduces *thresholded-rewards MDPs* (TRMDPs). We present an algorithm that efficiently computes the optimal policy for a TRMDP.
- **Chapter 4** introduces three heuristic techniques that allow us to derive approximate solutions to TRMDPs. We empirically measure the performance of these heuristics on some sample problems.
- **Chapter 5** introduces thresholded-rewards semi-MDPs (TRSMDPs), which allow us to model domains in which the time needed for a state transition follows any arbitrary distribution.

- **Chapter 6** relaxes our previous assumption that the behavior of the opponent is known to us *a priori*. We assume instead that we have some means of recognizing the opponent's behavior online. This allows our team to change strategy based on the recognized behavior of the opponent.

- **Chapter 7** extends TRMDPs to domains with unknown rewards. In this chapter, we provide a sampling-based control algorithm which assumes that our agent occasionally receives samples of the reward received.

- **Chapter 8** discusses lines of related research, including MDPs and semi-MDPs, decision problems with alternative objective functions, multi-robot teamwork, and previous approaches to team strategy in the robot soccer domain.

- **Chapter 9** summarizes our major findings and suggests potential lines of future research.

- **Appendix A** provides details on the communication strategies and the play-based role assignment algorithms used by our AIBO robot soccer team during the 2004–2008 RoboCup competitions.

- **Appendix B** presents the full results of our experiments in the Capture the Flag domain.

Chapter 2

Domains

In this chapter, we describe the three experimental domains that are used throughout this thesis: robot soccer, Capture the Flag, and reCAPTCHA. These domains all feature a hard time deadline, some notion of score, and an overall goal of attaining a certain score threshold by the end of the time horizon.

In *robot soccer*, two teams of robots play a twenty-minute game of soccer; the goal is to be ahead in score when time runs out. Robot soccer is the domain which originally inspired this thesis: I have been involved with CMU's CMPack and CMDash robot soccer teams since 2003, particularly focusing on teamwork and multi-robot coordination. Robot soccer is a challenging domain for many reasons, including: highly stochastic actions, limited and noisy perception of the environment, distributed teams (requiring the use of wireless networking for communication and coordination), and the presence of adversaries in the environment. We have addressed many of these challenges in earlier work [37–39,75], but the primary focus of this dissertation is on the adversaries: how can we act so as to maximize the probability of beating the opponent? Section 2.1 describes the robot soccer domain in detail.

Capture the Flag (CTF) is another timed, zero-sum game in which the goal is to be ahead of the opponent when the game is over. In CTF, each team possesses a flag and attempts to capture the opponents' flag while simultaneously defending its own flag from capture by the opponents. We have developed a grid-world CTF simulation that eliminates many of the complexities of robot soccer, such as highly stochastic actions, partial observability, and the need for communication. By eliminating these complexities, we can focus solely on the challenges that are most relevant to this thesis: finding team strategies that aim to win against the opposing team. Section 2.2 describes the CTF domain in detail.

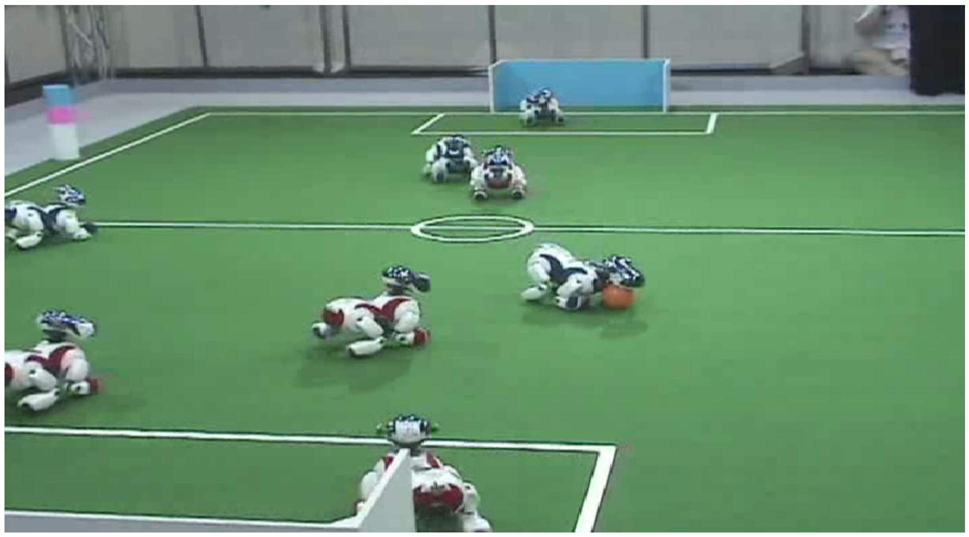


Figure 2.1: A RoboCup four-legged league soccer match from 2005.

reCAPTCHA is a human computation project that digitizes books by enlisting humans to read words that are difficult for OCR algorithms to read. In the reCAPTCHA domain, we are given a document that needs to be digitized by some time deadline, and need to act so as to maximize the probability that the document is fully digitized by the deadline. Unlike robot soccer and CTF, reCAPTCHA is not an adversarial domain; there is no opponent we are trying to defeat. However, there is still a notion of "score" (the number of words successfully digitized so far) and time, and the overall goal is to attain a certain score threshold (successfully digitizing every word in the document) by the deadline. An additional challenge in the reCAPTCHA domain is *unknown rewards*. Since reCAPTCHA does not know whether a human's answer for a word is correct or not, rewards are hidden from the agent at execution time. Section 2.3 describes the reCAPTCHA domain in detail.

2.1 Robot Soccer

Since 2003, I have worked on multi-robot coordination as part of the CMPack and CMDash entries to the RoboCup four-legged league [75]. In this domain, two teams of Sony AIBO robots play each other in a game of soccer [24, 25]. The general features of the domain are as follows:

- Each team has four robots: typically, three field players and a goalkeeper. (In 2008, the team size was expanded to five robots.)

- A game consists of two ten-minute-long halves.
- The field's size has ranged from 4.2m × 2.7m (in 2004) to 6m × 4m (in 2005–2007) to 7.5m × 5m (in 2008).

The exact rules of the game and the setup of the environment are changed every year in order to create new research challenges. The complete rules of the RoboCup four-legged league for the 2004–2008 seasons are available online [52–56]. Throughout this thesis, whenever robot soccer experiments are described, we will refer to the specific set of rules that were utilized in the experiment. Figure 2.1 shows a snapshot of a RoboCup four-legged league soccer match from 2005.

There are many features of the RoboCup domain that make it a rich and challenging testbed for multi-robot team coordination. These features include:

- **Full autonomy:** each team of robots operates completely without human supervision. However, teams are allowed to change the robots' programming at halftime or during a timeout. Each team is granted one timeout per game.
- **Distributed teams:** all perception, computation, and action is done on-board the robots. The robots are equipped with 802.11b wireless networking, which enables communication among team members; however, the robots are not allowed to communicate with any off-board computers.
- **Limited perception:** each robot's primary sensor is a low-resolution camera (208 × 160 pixels) with a very narrow field of view (under 60 degrees). A single robot therefore has a very limited view of the world, so teams can benefit greatly from communication strategies that build a shared world model.
- **Dynamic, adversarial environment:** the presence of adversaries in the environment is a significant challenge. Opponents ensure that the environment is extremely dynamic: within a few seconds, the state of the world may change significantly.
- **Temporal constraints:** there are two temporal constraints that arise due to the presence of adversaries. First, all team decisions must occur in real time. A team that takes too long to coordinate will have robots that display hesitation in carrying out its tasks, which gives the opponents a significant advantage. Second, soccer is a finite-horizon zero-sum game. A game of soccer has a winning team, a losing team, and a defined ending point. Playing a conservative strategy—which might work well

over a long period of time—is of no use to a team that is losing and only has a few seconds remaining in the game. A team in this situation must choose a strategy that can score a goal quickly, even if such a strategy has other weaknesses. Multiple team coordination strategies are needed.

- **High network latency:** the presence of dozens of robots in the competition environment leads to very unpredictable quality of the robots' wireless network. Teams may experience periods of high network latency and collisions; latencies of over a second have been observed. To achieve consistent performance, a team needs to ensure that the coordination strategies employed are robust to disruptions in communication.

2.1.1 Play-Based Teamwork in Robot Soccer

In general, multi-agent teamwork consists of a team control policy, i.e., a selection of a joint action by teammates given a perceived state of the environment [47,70]. It is our experience that it is rather challenging to generate or learn a team control policy in complex, highly dynamic (in particular adversarial), multi-robot domains. Therefore, instead of approaching teamwork in terms of a mapping between state and joint actions, we follow a *play-based* approach, as introduced by Bowling et al. [5,6,8]. Plays allow a team of robots to discretely change their strategy depending on the state of the game. The inspiration for this thesis is the desire for *optimal play selection* in a robot soccer team, where the optimal play choice (at any point in time) is that which is most likely to lead to a victory.

In [37], we empirically show the need for switching between two different ways of handling a simple robot soccer scenario, depending on the behavior of an opponent robot. We argue that the use of high-level coordination strategies is most appropriate for the RoboCup four-legged league, due to the distributed, dynamic, adversarial environment and the presence of high network latency.

Encouraged by this initial result, we pursued a *play-based* coordination strategy for the 2005 RoboCup international competition [38,39]. A *play* is a team plan that provides a set of *roles*, which are assigned to the robots upon initiation of the play. Each role is a top-level individual behavior, such as Defender, Midfielder, or Striker, that directs the actions of a single robot. Table 2.1 summarizes the roles used by our team in the 2005 competition. Bowling et al. [5] introduced a play-based method for team coordination in the RoboCup small-size league. However, the small-size league has centralized control of the robots. We have extended this work by introducing a distributed, play-based approach to teamwork [38,39]. Our play system is the first known implementation of play-based teamwork in a distributed team of robots. This play-based approach allows our team to handle the various challenges of the

Role	Region	Brief Description
`Goalkeeper`	In front of the goal	Blocks opponent shots on goal and clears the ball.
`Independent`	Entire field	Chases the ball anywhere on the field and attempts to score goals.
`Defender`	Defensive half of the field	Clears the ball and harasses opposing attackers.
`Striker`	Offensive half of the field	Plays in the front half of the field and attempts to score goals.
`Midfielder`	Near the midfield line	Behaves like a Defender but covers an area further forward.
`LeftFlanker`	Left side of the field	Runs the entire length of the field to defend or attack, but remains on only the left half of the field.
`RightFlanker`	Right side of the field	Runs the entire length of the field to defend or attack, but remains on only the right half of the field.
`MidfieldDefender`	Midfield and defensive half	Behaves like a defender but covers the entire defensive half and the area near midfield.
`KickoffStriker`	Near the center circle	Special role used to perform a kickoff.
`KickoffCharger`	On the midfield line	Special role used to follow up on a kickoff.

Table 2.1: Summary of the roles used by our AIBO robot soccer team in RoboCup 2005.

RoboCup domain listed above. Full details of our play-based approach are presented in Appendix A; in this section, we only discuss the features directly relevant to our focus on high-level team strategies.

Multiple plays allow us to capture different teamwork strategies, as explicit responses to different types of opponents. Plays also allow the team to reason about the zero-sum, finite-horizon aspects of the RoboCup domain: the team can change plays as a function of the score and time left in the game.

We have designed a language for specifying plays, which is inspired by the work of Bowling et al. Our language allows us to define *applicability conditions*, which denote when a play is suitable for execution and which *roles* should be assigned to the robots. Since robots

```
PLAY Defensive
APPLICABLE fewerPlayers
APPLICABLE secondHalf winningBy2OrMoreGoals
ROLES 1 Goalkeeper
ROLES 2 Goalkeeper Defender
ROLES 3 Goalkeeper Defender Independent
ROLES 4 Goalkeeper Defender Midfielder Independent
```

Figure 2.2: Definition of the `Defensive` play.

occasionally crash, and are temporarily removed from play due to committing fouls, each play also specifies which roles to assign in the event that the team is not playing at full strength.

Applicability. An applicability condition denotes when a play is suitable for execution. Each applicability condition is a conjunction of binary predicates. A play may specify multiple applicability conditions; in this case, the play is considered executable if any of the separate applicability conditions are satisfied.

Roles. Each play specifies which roles should be assigned to a team with a variable number of robots by defining different `ROLES` directives. A directive applies when a team has k active robots, and specifies the corresponding k roles to be assigned. If a robot team has n members, each play has a maximum of n `ROLES` directives. Since our AIBO team is composed of four robots, our plays have four `ROLES` directives.

Unlike the work of Bowling, we do not have `DONE` or `TIMEOUT` keywords that specify when a play is complete. Rather, a play is considered to be complete as soon as the play selector chooses a different play, which may happen because the current play's applicability conditions are no longer met or because the team has determined that some other play is preferable.

Figure 2.2 shows an example of a defensive play. Its applicability conditions specify that this play is applicable 1) when our team has fewer active players than the opponents or 2) when the game is in the second half and our team is winning by at least two points. If all four of our robots are active, `Defensive` assigns the roles Goalkeeper, Defender, Midfielder, and Independent; if only three robots are active, `Defensive` assigns the roles Goalkeeper, Defender, and Independent; and so on. Table 2.2 summarizes the seven plays used by our team in the 2005 competition.

A *play selector* algorithm runs continuously on one robot that is arbitrarily chosen to be

Play	Roles assigned
`Default`	`Goalkeeper Defender Striker Independent`
`Defensive`	`Goalkeeper Defender Midfielder Independent`
`Guard`	`Goalkeeper Defender MidfieldDefender Independent`
`Flankers`	`Goalkeeper Defender LeftFlanker RightFlanker`
`Aggressive`	`Goalkeeper LeftFlanker RightFlanker Independent`
`PullGoalie`	`Midfielder LeftFlanker RightFlanker Independent`
`Kickoff`	`Goalkeeper Defender KickoffCharger KickoffStriker`

Table 2.2: Summary of the seven plays used by our AIBO robot soccer team in RoboCup 2005.

the leader. The play selector chooses which play the team should be running. The leader periodically broadcasts the current play to its teammates, who set their roles (top-level behaviors) based on what the play specifies. In our initial approach [39], each play had an associated weight value, and the play selector would always choose the highest-weight play from the set of applicable plays. This allowed us to hand-code a strategy that depended on score and time, by utilizing score-based and time-based predicates (such as `secondHalf` and `winningBy2OrMoreGoals`) and setting play weights appropriately. This hand-coded approach was successful in the RoboCup 2005 international competition. However, all the reasoning about time and score was manually hand-coded into the plays. Instead, we would like the team to reason optimally about play selections, selecting the plays which are most likely to lead to a win. The work presented in this thesis aims to effectively address the problem of optimal play selection in domains such as robot soccer.

2.2 Capture the Flag

Capture the Flag (CTF) is a traditional game, played by two teams, in which each team possesses a *flag*, and attempts to capture the opponents' flag while simultaneously defending its own flag from capture by the opponents. Each time a flag is successfully captured, the capturing team scores a point. Like robot soccer, CTF is interesting to us because it is a timed, adversarial domain: the overall goal of each team is to be ahead in score when the game ends, after a given period of time has elapsed. AI researchers have studied CTF in many contexts, including video games and mixed human/robot teams [82]. There is no canonical version of the CTF domain. We have written our own multi-agent CTF simulator; we utilize this simulator for all the CTF experiments presented in this document. The source code for our simulator is available online [36]. In Section 2.2.1, we discuss the specification of the CTF domain; in Sections 2.2.2 and 2.2.3, we discuss the roles (top-level single-agent behaviors) available to each player and the plays (high-level team policies) available to each team.

2.2.1 Capture the Flag Domain Specification

A game of Capture the Flag takes place in a grid world $m \times n$ squares in size. There are two teams, Blue and Red, each consisting of the same number of players. The world is split into two halves, also known as *home zones*. Blue's home zone is on the left side of the world; Red's home zone is on the right. At the beginning of the game, each team's flag starts at a fixed location in its home zone; the players are initially positioned at random locations within their home zone. To score a point for its team, a player must *move* into the opposing home zone, *pick up* the opponents' flag, and *capture* the flag by carrying it back to its own home zone. To defend the flag, players can *tag* any nearby opponent that has entered the defenders' home zone. A player that has been successfully tagged is immediately teleported to the very back of its own home zone. Other than this loss of position, there is no additional penalty for being tagged. A CTF game lasts a fixed number of time steps. At the end of the game, whichever team has accumulated the most points (that is, successfully captured the opponent's flag the most times) is declared the winner. If both teams have the same number of points, the game ends in a draw.

In the experiments presented in this thesis, we use a 72×48 world, five players per team, and a time horizon of 2000 time steps. All of these constants were chosen such that the CTF game has roughly similar characteristics to (human or robot) soccer games. The world size is proportional to the 6m $\times$ 4m robot soccer field, and the choice of time horizon means that the average final score for each team is similar to soccer, with each side usually scoring approximately 0–5 points in a game. The choice of five players per team gives teams enough players to allow a variety of different team-level strategies to be employed. Five players is also small enough that the effect of changing a single player's role can lead to a significant difference in the overall outcome.

Figure 2.3 shows a screenshot of our Capture the Flag simulator (scaled down to half the usual size: 32×24). Each player is depicted as a colored circle with a number written on it; the flags are depicted as colored squares. The blue and red home zones are the colored regions on the left and right sides of the picture (respectively). Two blue players have taken defensive positions near their flag on the left side of the world. Over on the right, three red players are attempting to defend the red flag from two blue attackers. The remaining players

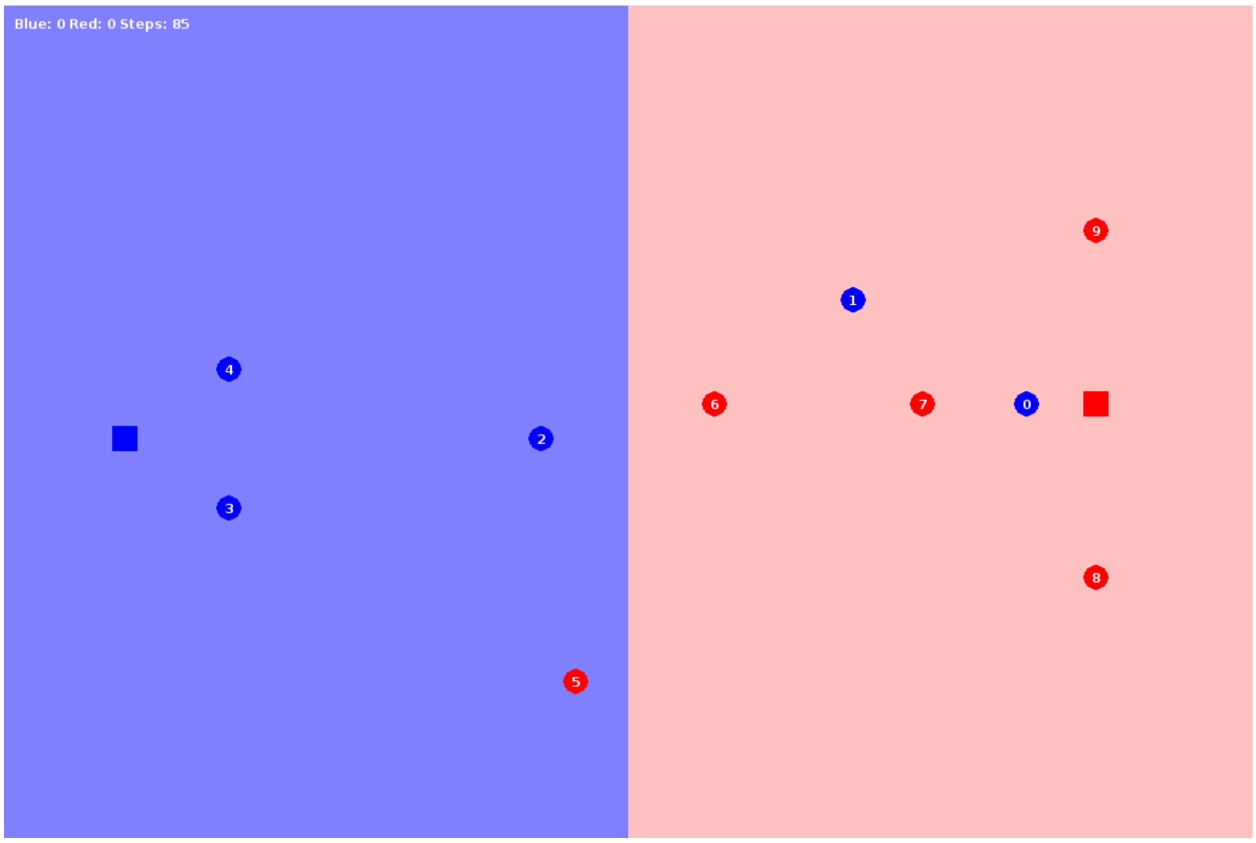


Figure 2.3: A screenshot of our Capture the Flag simulator. Each player is depicted as a colored circle with a number written on it; the flags are depicted as colored squares.

(one blue and two red) are currently near midfield.

CTF World State

There are 12 domain objects in the CTF domain: 5 blue players $p^b_{0\ldots4}$, 5 red players $p^r_{0\ldots4}$ and 2 flags f^b and f^r. Each player p_i is fully described by its position on the field (x_{p_i}, y_{p_i}). Players cannot share the same location; if a player attempts to move into a square already containing a player, the move action will fail. Each flag f^k is defined by its position (x_{f^k}, y_{f^k}) and a value c^k that denotes which player is currently holding the flag, if any. If a player p_i is holding the flag f^k, c^k is set to i; otherwise, c^k is set to None, a special value which indicates that no player is holding the flag. For convenience, we define the set $P = \{p^b_{0\ldots4}\} \cup \{p^r_{0\ldots4}\}$ of all players and the set $F = \{f^b, f^r\}$ of flags.

Though the world state can be fully described with only 26 variables (the x- and y-positions of each of the 12 objects, plus the c^k values of each flag), the total number of possible states in the domain is extremely large. Since the world is 72×48 squares in size, each of the 10 players can be in one of 3456 possible locations, though no two players can share the same location. Each flag can only be in one-half of the field (otherwise a score would occur), so there are 1728 possible locations for each flag. This leads to a lower bound of $\left(\prod_{i=0}^{9} 3456 - i\right) \times 1728^2 \approx 7.16 \times 10^{41}$ states. (The actual number of world states is actually slightly higher than this since the above calculation doesn't consider the c^k value for either flag.)

CTF Actions and Effects

There are four types of low-level actions that can be chosen by each player in the CTF domain. Players can *move* in the four cardinal directions (north, south, east, or west), *stay* in position, *pick up* the opposing team's flag, and *tag* opposing players. Table 2.3 provides a brief description of each action, along with which roles use that action. (Roles are the top-level behaviors available to each player; CTF roles are described in full detail in Section 2.2.2.) The full effects of these actions are described below and also defined formally in Algorithms 2.1, 2.2, 2.3, and 2.4.

- `MoveAction`(d). (Formally described in Algorithm 2.1.) A player can choose to move in any of the four cardinal directions. If the move action succeeds, the player will move one square in the appropriate direction (lines 2–9 and 24–25). However, move actions

Action	Brief Description	Used By
`MoveAction`(d)	Move one square in direction d. (Stochastic and conditional)	Attacker, Midfielder, Defender
`StayAction`	Stay in the same location. (Deterministic)	Midfielder, Defender
`PickupAction`	Pick up the opponents' flag. (Conditional)	Attacker
`TagAction`(p_t)	Tag an adjacent opponent p_t. (Conditional)	Attacker, Midfielder, Defender

Table 2.3: Summary of the low-level CTF actions available to each player. Some of the low-level actions are only utilized by certain player roles.

fail stochastically: with probability 0.1, the player's position remains unchanged (lines 11–12). Move actions also fail if the player requested to move outside the boundaries of the world (lines 13–14), or into a square already occupied by some other player (lines 15–17). Additionally, a player is not allowed to move within a certain distance of its own flag unless the flag is being held by an opposing player (lines 18–23). This is needed in order to give attacking players a chance to escape with the flag; otherwise a team could mount a perfect defense by positioning four defenders immediately adjacent to the flag. We found that a minimum distance of 5 squares was sufficient to allow a reasonable tradeoff between the ability of attackers to pick up the flag and the ability of defenders to successfully defend the flag from attackers. A player that is already within Manhattan distance 5 of its own flag (perhaps due to successfully tagging an opponent that had been holding the flag) is not allowed to move closer to the flag. Such a player can move further away from the flag. If a player is carrying the opponents' flag and moves successfully, the flag moves to the same square as the player (lines 26–28). If the new flag location is in the player's home zone, that player's team scores a point (lines 29–30).

- `StayAction`. (Formally described in Algorithm 2.2.) A player can choose to stay in the same position. This action always succeeds; the state of the world remains unchanged.

- `PickupAction`. (Formally described in Algorithm 2.3.) A player can choose to pick up the opposing team's flag. This action is legal only if the player's position is equal to the flag's position and the flag's c^k value is None (lines 2–3). The result of this action is that the c^k value of the flag is set to the ID number of the player (line 4). Any future move actions by this player will cause the flag to move along with the player until either a score occurs or an opponent successfully tags the player. A player is not allowed to pick up its own team's flag.

- `TagAction`(p_t). (Formally described in Algorithm 2.4.) A player can choose to tag a specific opponent p_t. A tag action succeeds only if the target is adjacent to the tagging player and in the tagging player's home zone (lines 2–3). If the tag is successful, the opponent is immediately teleported to a position at the back of the opponents' home zone (line 4). If the target was holding a flag, the flag's c^k value is set to None (lines 5–7). If the tag is unsuccessful, there is no effect on the target player; the tagging player remains in the same position.

In a single timestep of the CTF domain, all agents simultaneously observe the state of the world and choose individual actions. Since CTF is a multi-agent domain, conflicting actions can occur; for example, two agents might both attempt to move into the same square. In order to resolve any possible conflicts, players' actions are applied in a random order. If one player's action becomes invalid due to the previous action of another player, the new action has no effect. The ordering of actions is particularly important when considering the effects of tag actions; if the target of a tag action moves before the tag action is applied, the tag action will fail since the target is no longer adjacent to the tagging player. Therefore, the success rate of a given tag action is expected to be around 50%, assuming that the two players were initially adjacent and that the target chooses to move in a direction away from the tagging player.

2.2.2 Roles

We have developed three roles (top-level player behaviors) for our CTF domain: *attacker*, *defender*, and *midfielder*. At each time step the role is given the world state (as specified above) and chooses an appropriate action. The behavior of each of these roles is described below. We present formal descriptions of the behavior of each of the roles after the descriptions.

- **Attacker.** The Attacker role is primarily responsible for entering the opponents' half of the field and scoring. Each attacker attempts to approach the opponents' flag, pick it up, and bring it back to the home zone to score. Attackers are almost entirely single-minded in their pursuit of the flag; upon being tagged, an attacker immediately heads back toward the opposing flag. If an attacker is in its home zone, it will not go out of its way to tag an opposing player; however, if the attacker happens to be directly adjacent to an opposing player, the attacker attempts to tag the opponent. Algorithm 2.5 formally describes the behavior of the attacker role.

Algorithm 2.1 Effects of a `MoveAction` in the CTF domain.

```
1: Given: player $p_i^k$ playing for team $k$, direction $d \in \{N, S, E, W\}$
2: if $d = N$ then
3:     $(x, y) \leftarrow (x_{p_i}, y_{p_i} - 1)$
4: else if $d = S$ then
5:     $(x, y) \leftarrow (x_{p_i}, y_{p_i} + 1)$
6: else if $d = E$ then
7:     $(x, y) \leftarrow (x_{p_i} - 1, y_{p_i})$
8: else if $d = W$ then
9:     $(x, y) \leftarrow (x_{p_i} + 1, y_{p_i})$
10: $m \leftarrow$ True    // if set to False, the move action fails
11: if RANDOM() $< 0.1$ then
12:     $m \leftarrow$ False
13: if INVALID-POSITION$(x, y)$ then
14:     $m \leftarrow$ False
15: for $p_j \in P$ do
16:     if $i \neq j \wedge (x, y) = (x_{p_j}, y_{p_j})$ then
17:         $m \leftarrow$ False
18: for $f^j \in F$ do
19:     if $j \neq k \wedge c^j =$ None then
20:         $\delta =$ MANHATTAN$((x_{f^j}, y_{f^j}), (x_{p_i}, y_{p_i}))$
21:         $\delta' =$ MANHATTAN$((x_{f^j}, y_{f^j}), (x, y))$
22:         if $\delta' < 5 \wedge \delta' < \delta$ then
23:             $m \leftarrow$ False
24: if $m =$ True then
25:     $(x'_{p_i}, y'_{p_i}) \leftarrow (x, y)$
26:     for $f^j \in F$ do
27:         if $c^j = i$ then
28:             $(x'_{f^j}, y'_{f^j}) \leftarrow (x, y)$
29:             if $(x, y)$ is in $p_i$'s home zone then
30:                 $\text{score}^{k\prime} \leftarrow \text{score}^k$
31: else
32:     $(x'_{p_i}, y'_{p_i}) \leftarrow (x_{p_i}, y_{p_i})$
```

Algorithm 2.2 Effects of a `StayAction` in the CTF domain.

```
1: Given: player $p_i^k$ playing for team $k$
2: $(x'_{p_i}, y'_{p_i}) \leftarrow (x_{p_i}, y_{p_i})$
```

Algorithm 2.3 Effects of a `PickupAction` in the CTF domain.

```
1: Given: player $p_i^k$ playing for team $k$
2: for $f^j \in F$ do
3:     if $j \neq k \land c^j =$ None $\land (x_{p_i}, y_{p_i}) = (x_{f^j}, y_{f^j})$ then
4:         $c^{j'} \leftarrow i$
```

Algorithm 2.4 Effects of a `TagAction` in the CTF domain.

```
1: Given: player $p_i^k$ playing for team $k$, target player $p_t^\kappa$ (with $k \neq \kappa$)
2: $\delta \leftarrow$ Manhattan$((x_{p_i}, y_{p_i}), (x_{p_t}, y_{p_t}))$
3: if $\delta = 1 \land p_t$ is in $p_i$'s home zone then
4:     $(x'_{p_t}, y'_{p_t}) \leftarrow$ (random position at the back of $p_t$'s home zone)
5:     for $f^j \in F$ do
6:         if $c^j = t$ then
7:             $c^{j'} \leftarrow$ None
```

- **Defender.** The Defender role is tasked with defending the flag from capture by the opponent team. A team's defenders encircle the flag (at the minimum legal radius of 5 squares) and wait for the opponents' attackers to approach. The exact positions taken by the defenders depend on the number of players playing in the Defender role. If only one player is defending, it will take a position between the flag and the opponents' home zone; if four players are defending, they will encircle the flag from all sides. Figure 2.4 shows the exact positions taken by the defenders in each possible case. If any opponents are near enough to tag, the defender attempts to tag them, preferring to tag a player which is holding the flag (if any). If any opponent possesses the flag but is not near enough to be tagged, the defender moves toward the opponent in an attempt to get within tagging distance. If an opponent possessing the flag is tagged, the defenders re-form their circle around the new location of the flag. Algorithm 2.6 formally describes the behavior of the defender role.

- **Midfielder.** The Midfielder role is a primarily used as a second line of defense in the event that an opposing attacker manages to pick up the flag and escape the encircling defenders. The midfielder positions itself in the home zone, near the midfield line and between the team's flag and the opponents' home zone. The midfielder waits in position until the team's flag is picked up by an opponent. Once the flag is picked up, the midfielder moves toward the opponent and attempts to tag. If the tag is successful, the flag is dropped and the midfielder returns to its home position. The midfielder may also opportunistically tag opponents without the flag which happen to come nearby; however, the midfielder will not move far from its home position in order to give chase. Algorithm 2.7 formally describes the behavior of the midfielder role.

Formal descriptions of all three roles are provided below. We assume the existence of a library of basic low-level functions:

- GO-TO-POINT(x, y) returns a `MoveAction` that moves the player one square closer to the destination (x, y). It is called by all the roles in order to move the players to their desired positions.
- CHOOSE-TAG-TARGET() returns the ID of an opponent that can be tagged, or None if there are no opponents near enough to tag. If multiple opponents are adjacent, this function returns the one which is carrying the flag, if any; otherwise a random opponent is returned. This function is called by all the roles in order to tag opponents.
- CHOOSE-RETURN-LOCATION() chooses a random location in the player's home zone, near the midfield line. It is called by the attacker when it picks up the opponents' flag. As long as the attacker continues to carry the flag, the attacker will move towards the return location chosen, in order to successfully complete the capture of the opponents' flag.
- CHOOSE-DEFENSE-POSITION() chooses the home position used by the defender. As shown in Figure 2.4, the home position of each defender depends on the number of defenders assigned by the current play.
- CHOOSE-MIDFIELD-POSITION() chooses the home position used by the midfielder.

Algorithm 2.5 Behavior of the CTF Attacker role.

```
1: Given: player $p_i^k$ playing for team $k$
2: $j \leftarrow$ OTHER-COLOR($k$)
3: if $(x_p, y_p) = (x_{f^j}, y_{f^j})$ then
4:     if $c^j =$ None then
5:         $\rho \leftarrow$ CHOOSE-RETURN-LOCATION()
6:         return PickupAction()
7:     else if $c^j = i$ then
8:         return GO-TO-POINT($\rho$)
9: $\tau \leftarrow$ CHOOSE-TAG-TARGET()
10: if $\tau \neq$ None then
11:     return TagAction($\tau$)
12: return GO-TO-POINT($x_{f^j}, y_{f^j}$)
```

Algorithm 2.6 Behavior of the CTF Defender role.

```
1: **Given:** player $p_i^k$ playing for team $k$
2: $\tau \leftarrow$ Choose-Tag-Target()
3: if $\tau \neq$ None then
4:     return TagAction($\tau$)
5: if $c^k =$ None then
6:     return Go-To-Point(Choose-Defense-Position())
7: else
8:     return Go-To-Point($x_{f^k}, y_{f^k}$)
```

Algorithm 2.7 Behavior of the CTF Midfielder role.

```
1: **Given:** player $p_i^k$ playing for team $k$
2: $\tau \leftarrow$ Choose-Tag-Target()
3: if $\tau \neq$ None then
4:     return TagAction($\tau$)
5: if $c^k =$ None then
6:     return Go-To-Point(Choose-Midfield-Position())
7: else
8:     return Go-To-Point($x_{f^k}, y_{f^k}$)
```

2.2.3 Plays

As in robot soccer, each play in CTF provides an assignment of roles to each player on the team. We name each play in the form: Aa_Mm_Dd, where a, m, and d are the number of attackers, midfielders, and defenders on the team, respectively. For example, the play A3_M1_D1 assigns three attackers, one midfielder, and one defender. We found that having more than one midfielder on the team was never significantly better than having zero or one midfielders. For the work presented in this thesis, we therefore restricted the set of plays to all those which included zero or one midfielder. There are eleven such plays: six plays with no midfielder (ranging from zero to five attackers on the team, with the remainder of the players defending), and five plays with one midfielder (ranging from zero to four attackers, with the remainder of the players defending).

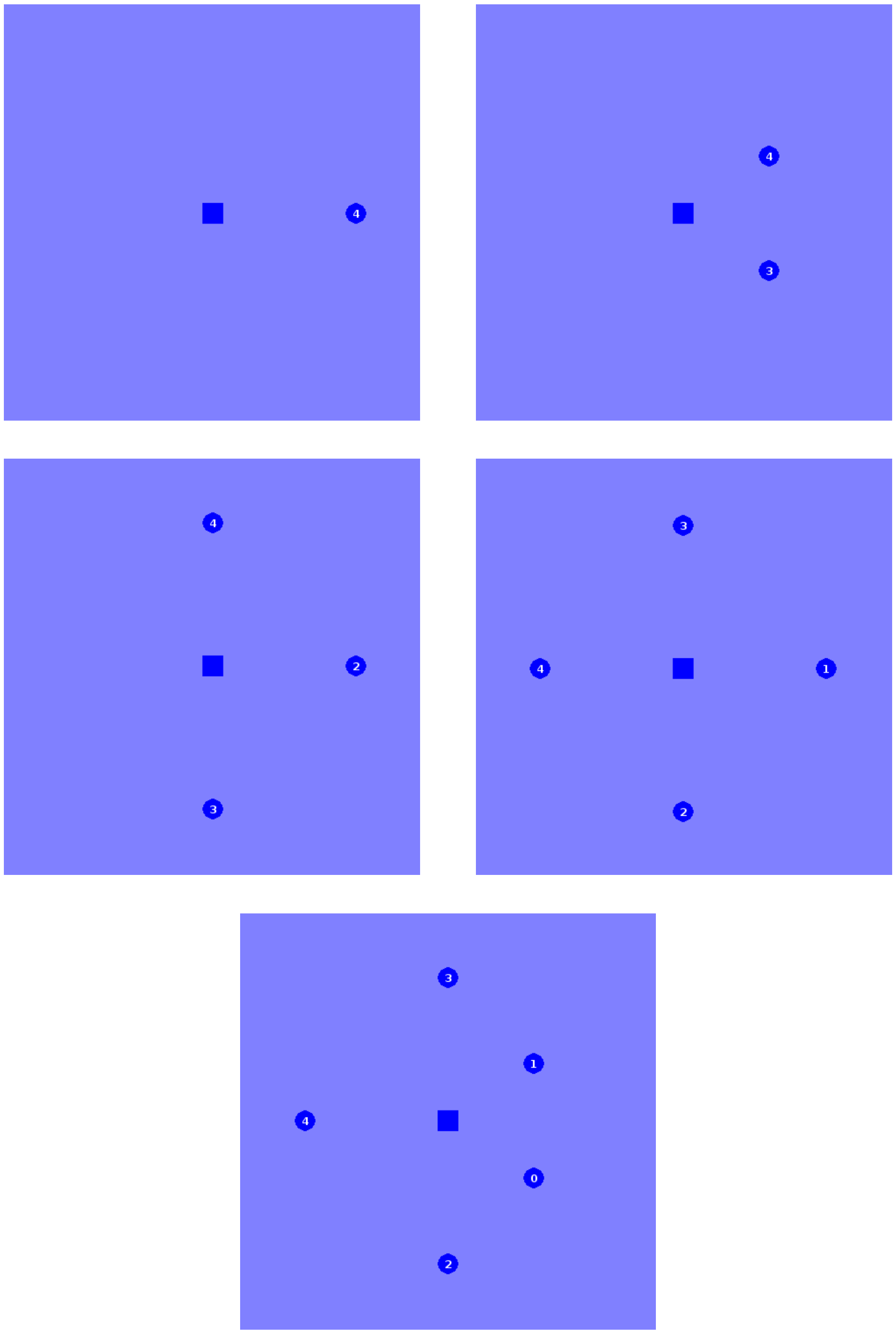


Figure 2.4: Positions taken by the defenders depending on the number of defenders on the team. The opponents' home zone is to the right.

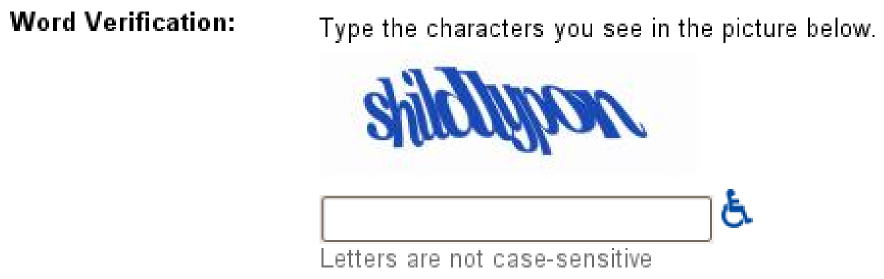

Word Verification:
Type the characters you see in the picture below.
shilthypon
Letters are not case-sensitive

Figure 2.5: A sample CAPTCHA, shown to a user as part of the account signup process for creating a Google Mail (Gmail) account.

2.3 reCAPTCHA

A CAPTCHA[1] is a challenge-response test that can be used to tell humans and computers apart [76]. CAPTCHAs are typically used to prevent automated registrations on Web-based email services and other Web sites. The most common form of CAPTCHA consists of an image that contains distorted words or letters, as shown in Figure 2.5. The user is prompted to type in the letters that appear in the image. In order to correctly distinguish between humans and computers, a CAPTCHA must present the user with a challenge that is relatively easy for humans to solve, but difficult or impossible for computers to solve. Therefore, every time a human solves a CAPTCHA, he/she is performing a task that is known to be difficult for state-of-the-art AI algorithms.

The reCAPTCHA project, `http://recaptcha.net`, aims to make positive use of this precious *human computation* power by using human responses to CAPTCHAs to aid in the digitization of old books, newspapers, and other texts [77]. Typically, such texts are digitized by scanning the entire source document, then running optical character recognition (OCR) software on the resulting images in order to recover the original text in digital form. However, state-of-the-art OCR software cannot achieve perfect transcription accuracy, especially on old books in which the words are often faded and distorted. In these cases, the OCR software will output its best guess for the word, but with a low confidence score. reCAPTCHA takes these low-confidence OCR words and displays them to users in the form of a CAPTCHA challenge. These human answers are then used to improve the accuracy of the digitized text. Figure 2.6 shows an example of this process. In this example, the circled word is read as "announcacl" by the OCR software, but with low confidence. This *unknown word* is extracted from the source document and shown to users, who type in the correct spelling: "announced." To check whether the user is a human, the CAPTCHA also displays

[1]The word "CAPTCHA" is an acronym for Completely Automated Public Turing test to tell Computers and Humans Apart.

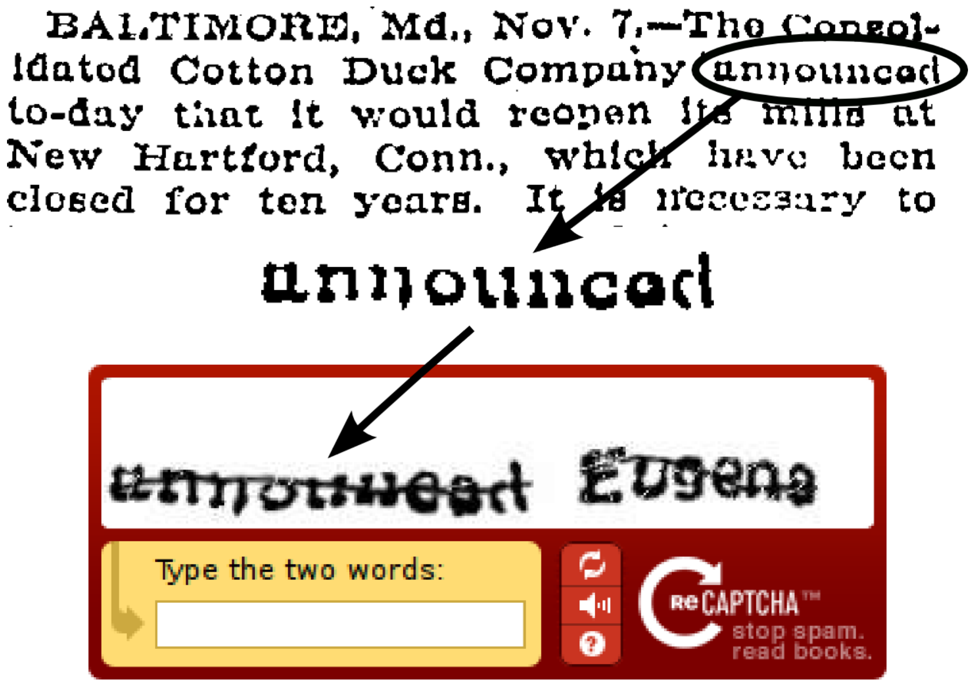

BALTIMORE, Md., Nov. 7.—The Consol-
idated Cotton Duck Company announced
to-day that it would reopen its mills at
New Hartford, Conn., which have been
closed for ten years. It is necessary to

Figure 2.6: reCAPTCHA digitization example.

a *known word*, for which the correct spelling is already known—"Eugene" in the example above. To ensure that automated programs cannot read the words, both words are distorted, and a black line is drawn through them.

Figure 2.7 shows a simplified outline of the complete reCAPTCHA system. reCAPTCHA maintains pools of known words and unknown words. When a new document is added to the system, any words that OCR cannot confidently read are added to the pool of unknown words. For some documents, there may also be an associated time deadline; if so, the system only has a limited amount of time available to digitize the document. For the purpose of this thesis, the most important aspect of the reCAPTCHA system is the "Challenge Generator," which is responsible for choosing words from the word pools and displaying them to users in the form of CAPTCHA challenges. If we have a hard time deadline, the challenge generator is faced with a control problem: how should we choose challenges such that we maximize the probability of digitizing the entire document before the deadline? This turns out to be a particularly challenging problem because the rewards achieved are unknown; i.e., the system does not know whether a user's response to an unknown word is actually the correct spelling of that word. We address this problem of *unknown rewards* in Chapter 7.

During normal reCAPTCHA operation, the challenge generator sends one known word and

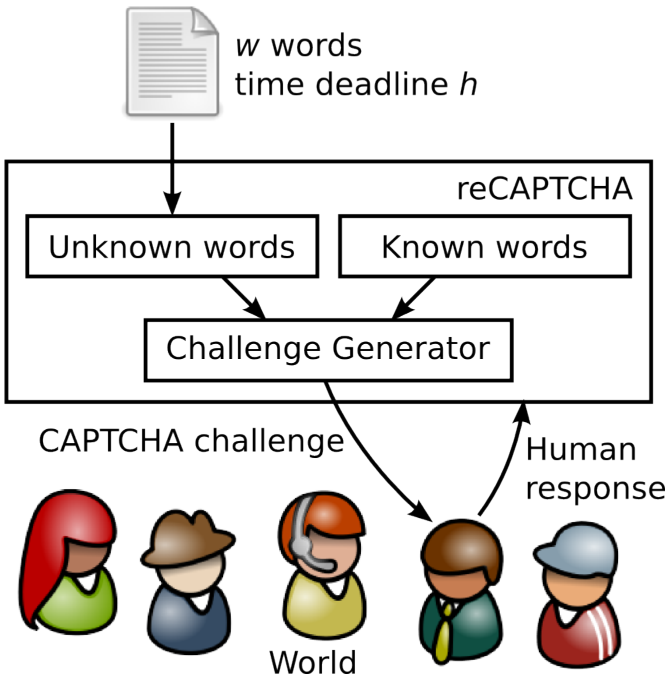


Figure 2.7: Outline of the reCAPTCHA system.

one unknown word to each user. If the user answers the known word correctly, their response for the unknown word is counted as a "vote" for the correct spelling of that word. If enough users agree on the spelling of the unknown word, it is moved into the pool of known words. More details on the production reCAPTCHA system, which has been online since May 2007, can be found in [77].

While most users make a good-faith effort to correctly transcribe the words, some users maliciously submit incorrect answers a high fraction of the time, which could potentially result in incorrect transcriptions. If a relatively large proportion of users are malicious at a given time, the challenge generator can limit the damage done by temporarily serving two known words to all users. This means that no new words are transcribed, but the quality of the transcription is not negatively affected by the malicious users. Conversely, if we know that all users are not malicious, the challenge generator could theoretically send two unknown words to each user, doubling the transcription rate but increasing the chance that any malicious user would be able to seed incorrect transcriptions into the system. (The production reCAPTCHA system would never send two unknown words to a user, because this would severely compromise the ability to distinguish humans from computers. However, for the experiments presented in this thesis, we only address the problem of digitizing documents, ignoring the security implications of such a choice.)

We present a detailed MDP model of the reCAPTCHA domain in Section 7.1. This model is derived by analyzing the answers provided by 31,163 of the most active reCAPTCHA users. Over a six-month period, these users submitted over 29 million answers to reCAPTCHA. By itself, this MDP model does not involve time or score. However, in Chapter 7, we address the problem of trying to maximize the probability that all w words in a document are successfully transcribed by some time deadline h. A further challenge in the reCAPTCHA domain is that the rewards achieved are unknown at execution time; i.e., the system does not know whether a user's response to an unknown word is actually the correct spelling of that word. We also address this problem of *unknown rewards* in Chapter 7.

2.4 Summary

In this chapter, we have presented the three domains that are used throughout this thesis: robot soccer, Capture the Flag, and reCAPTCHA. These domains all feature a hard time deadline, some notion of score, and an overall goal of attaining a certain score by the end of the time horizon. Robot soccer and Capture the Flag are timed, zero-sum games, in which the goal is to maximize our probability of *winning*: being ahead in score when time runs out. Robot soccer is a rich domain presenting many challenges; in this thesis, we focus mainly on the problem of choosing strategies such that we win against the opponent team. Capture the Flag eliminates many of the complexities of robot soccer, allowing us to focus solely on the challenges of finding team strategies that win against the opponent team. In the reCAPTCHA domain, we are given a document that needs to be digitized by some time deadline, and need to act so as to maximize the probability that the document is fully digitized by the deadline. Unlike robot soccer and Capture the Flag, reCAPTCHA is not an adversarial domain. However, we still care about the time remaining and the "score" (number of words successfully digitized so far). Furthermore, in the reCAPTCHA domain, we have *unknown rewards*: since we don't know whether a human's answer is correct or not, rewards are hidden from the agent at execution time.

In the next chapter, we introduce the concept of *thresholded-rewards MDPs* (TRMDPs). With TRMDPs, our true reward is calculated as a threshold function on the cumulative intermediate reward achieved during execution. For the domains presented in this chapter, these threshold functions enable us to accurately model the desired goals. For robot soccer and Capture the Flag, we want to maximize the probability that $\textit{our\_score} - \textit{opponent\_score}$ is positive at the end of the game; for reCAPTCHA, we want to maximize the probability that we have successfully digitized at least w words by the time deadline.

Chapter 3

Thresholded-Rewards MDPs

Markov Decision Processes (MDPs) are a powerful tool for planning in the presence of uncertainty. MDPs provide a theoretically sound means of achieving optimal rewards in uncertain domains. The standard MDP problem is to find a policy $\pi : S \rightarrow A$ that maps states to actions such that the cumulative long-term reward is maximized according to some objective function [4]. Over an infinite time horizon, the objective function is typically a sum of discounted rewards or the average reward rate as $t \rightarrow \infty$ [22, 34]. Over a finite time horizon, a discount factor is not needed, and the objective function is typically the sum of the rewards achieved at each time step.

As discussed in Chapter 2, our work is motivated by domains with a notion of "score" and limited time, including robot soccer, Capture the Flag (CTF), and reCAPTCHA. In timed, zero-sum games, such as robot soccer and CTF, winning against the opponent is more important than the final score. Therefore, a team that is losing near the end of the game should play aggressively to try to even the score even if an aggressive strategy allows the opponent to score more easily. Even in a domain without an adversary, such as reCAPTCHA (in which there is a given amount of work to complete in a finite amount of time), we might also expect that an agent which is "behind schedule" with respect to the deadline might act more aggressively, and that an agent which is ahead of schedule might act more conservatively. In Section 2.1, we discussed how a team of soccer-playing robots can change *plays* (high-level team strategies) based on factors such as the time remaining in a game and the score difference [38, 39, 61]. However, this strategy selection was hand-tuned, using simple rules such as, "If our team is losing and there is less than one minute remaining, play aggressively".

In this chapter, we utilize MDPs to derive optimal strategy selections for domains with score and limited time. Rather than maximizing the cumulative score over h time steps, we apply a threshold function f to the final cumulative score and seek to maximize the value of f. We call this the *thresholded-rewards* objective function. For zero-sum games, such as robot soccer, a thresholded-rewards objective function allows us to model our goal of *winning*: being ahead of the opponent after some number of time steps. The optimal policy for such a domain is one that maximizes the probability of being ahead at the end of the game. Such a policy will generally be nonstationary: the optimal action from a given state depends on the number of timesteps remaining and the current score difference.

In this chapter, we present an algorithm which takes in a *base MDP* describing the stationary dynamics of a domain and a *threshold function* we desire to maximize. This algorithm creates an expanded MDP which is annotated with score and time values; the resulting MDP is only polynomially larger than the base MDP. This expanded MDP can be solved using value iteration (or any other MDP solution technique) in order to recover the optimal policy. This optimal policy maximizes the expected value of the threshold function applied to the cumulative score. The running time of value iteration on the expanded MDP has a quadratic dependence on the number of states in the MDP and the length of the time horizon. For MDPs with large state spaces or long time horizons, the exact algorithm may be intractable. In the next chapter, we investigate a variety of approximate solution techniques for thresholded-rewards problems.

3.1 Definition of a Thresholded-Rewards MDP

We use the standard (S, A, T, R, s_0) notation for representing MDPs; for simplicity, we assume that rewards are found in the states of the MDP (rather than in state/action pairs). The optimal policy π of an MDP can be found exactly using a technique known as *value iteration*, which uses the Bellman equation [46]:

$$V^{n+1}(s) = \max_{a \in A} \Big\{ R(s) + \gamma \sum_{s' \in S} T(s, a, s') V^n(s') \Big\},$$

where $V^0(s) = R(s)$ and $\gamma \in (0, 1]$ is a *discount factor*. For an infinite-horizon problem, V^k converges to some V^* as $k \to \infty$ (for $\gamma < 1$). The optimal policy π^* for an infinite-horizon MDP is *stationary* (does not depend on time). For a finite-horizon problem with k timesteps remaining, V^k allows us to find the optimal next action from any state. This optimal action may depend on the number of time steps remaining; such a policy is said to be *nonstationary*. MDPs can also be solved with *policy iteration*. In this thesis, we primarily focus on value iteration techniques; however, the algorithms presented in this thesis can be trivially generalized to policy iteration techniques.

Let a *thresholded-rewards MDP* (TRMDP) be a tuple (M, f, h), where M is an MDP (S, A, T, R, s_0), f is a threshold function, and h is an integer (the time horizon). Informally, M runs for h time steps while the agent collects cumulative intermediate rewards $r_{intermediate}$; at the end, the agent receives a true reward r_{true} according to $f(r_{intermediate})$. A policy π for a TRMDP is nonstationary: it takes in a state $s \in S$, the time remaining, and the intermediate reward achieved so far. Formally, the dynamics of a TRMDP are as follows:

Algorithm 3.1 Dynamics of a thresholded-rewards MDP.

```
$s \leftarrow s_0$
$r_{intermediate} \leftarrow 0$
for $t \leftarrow h$ to 1 do
    $a \leftarrow \pi(s, t, r_{intermediate})$
    $s \leftarrow s' \sim T(s, a)$
    $r_{intermediate} \leftarrow r_{intermediate} + R(s)$
$r_{true} \leftarrow f(r_{intermediate})$
```

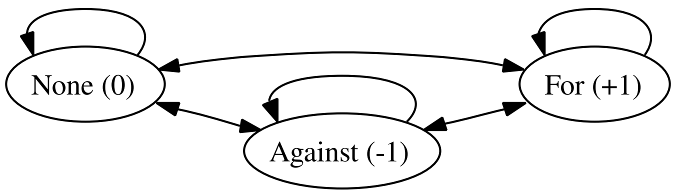
Robot soccer and CTF are timed, zero-sum games. In these domains, we consider the intermediate reward to be the difference between our agent's score and our opponent's score. We define the *zero-sum reward threshold function* as:

$$r_{true} = \begin{cases} 1 & \text{if } r_{intermediate} > 0 \\ 0 & \text{if } r_{intermediate} = 0 \\ -1 & \text{if } r_{intermediate} < 0. \end{cases} \tag{3.1}$$

This function assigns true reward of 1 for a win, -1 for a loss, and 0 for a tie. In the reCAPTCHA domain, we instead are given a document containing w words and want to ensure that we digitize all w words before the deadline. In this case, we need to use a *zero-one reward threshold function* such as:

$$r_{true} = \begin{cases} 1 & \text{if } r_{intermediate} \geq w \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

In a thresholded-rewards problem, we wish to find the optimal policy π^* that maximizes the expected value of r_{true}. It is important to note that, in general, 1) π^* will be nonstationary and 2) π^* is not the policy that maximizes expected intermediate reward. Though we focus on these two reward threshold functions in this thesis, our results generalize to arbitrary threshold functions.



a	$T(*, a,\text{For})$	$T(*, a,\text{Against})$	$T(*, a,\text{None})$
balanced	0.05	0.05	0.9
offensive	0.25	0.5	0.25
defensive	0.01	0.02	0.97

Figure 3.1: Example MDP M, inspired by robot soccer.

3.2 TRMDP Example

In a TRMDP, the optimal policy will generally be nonstationary. To illustrate this, we present an example—inspired by robot soccer—that will be used throughout the remainder of this chapter. We simplify the robot soccer domain significantly by modeling it as an MDP M with three states, as shown in Figure 3.1:

1. For: our team scores a goal (reward $+1$)
2. Against: the opponents score a goal (reward -1)
3. None: no score occurs (reward 0)

Our agent is a team of robots, and each action choice corresponds to a strategy (play) the team can adopt. In this example, we consider three different plays: *balanced*, *offensive*, and *defensive*. We treat the opponent team as static, using the transition probabilities to model the opponent as part of the environment. The transition probabilities of these actions are shown in Figure 3.1. When the *balanced* play is chosen, our agent has a 5% chance of scoring and the opponent has a 5% chance of scoring. The *offensive* play is risky: it increases our team's chance of scoring but gives the opponent an even greater chance of scoring. Inversely, the *defensive* play is conservative. For simplicity, these transition probabilities do not depend on the current state.

The expected one-step reward of action a from state s is equal to $\sum_{s'} R(s') \times T(s, a, s')$. Using this, we can compute the expected one-step reward of each action:

- *balanced*: $0 = 0.05 \times 1 + 0.05 \times -1 + 0.9 \times 0$
- *offensive*: $-0.25 = 0.25 \times 1 + 0.5 \times -1 + 0.25 \times 0$
- *defensive*: $-0.01 = 0.01 \times 1 + 0.02 \times -1 + 0.97 \times 0$

If we use the standard maximize-expected-rewards objective function, the optimal policy for M is to execute the *balanced* action at every time step. The *balanced* play has the highest expected one-step reward, and (for this MDP) also has the optimal expected long-term reward for any choice of γ in the range $[0, 1]$.

However, our team's goal is *not* to maximize expected rewards, but to maximize the probability that we finish the game with a higher score than the opponent. We view this as a thresholded-rewards problem: we apply the zero-sum threshold function to our agent's cumulative intermediate reward and maximize the expected value of this threshold function. With thresholded rewards, the policy of always choosing *balanced* has an expected true reward of 0, with the probability of winning being equal to the probability of losing. The exact probability of each result depends on the time horizon h and can be determined by the multinomial probability distribution. For $h = 120$, the probability of winning is 44.2%, the probability of losing is 44.2%, and the probability of tying is 11.6%.

However, this policy is suboptimal for the thresholded-rewards problem, as it is possible to achieve a positive true reward in this domain. In Section 3.5, we derive the optimal policy for this domain, which has an expected true reward of 0.1457 (for $h = 120$). Qualitatively, the optimal policy is nonstationary, choosing the *offensive* action if our agent is losing near the end of the game and choosing the *defensive* action if our agent is winning near the end of the game. By doing so, the optimal policy increases the chance of getting a "win" or "tie" result, at the expense of maximizing the expected value of the intermediate rewards.

3.3 TRMDP Conversion Algorithm

The MDP M shown in Figure 3.1 is a *base MDP*: M gives the transition dynamics of a domain, but M does not include any concept of cumulative score or time remaining. To solve the thresholded-rewards problem optimally, we need to include all possible combinations of base states, score, and time remaining.

In order to solve a TRMDP (M, f, h), we create a new MDP M' such that finding the policy that maximizes reward in M' is equivalent to finding the policy that maximizes $f(r_{intermediate})$

Algorithm 3.2 Converts a TRMDP (M, f, h) into an minimal MDP M' suitable for finding the optimal thresholded-rewards policy.

```
 1: Given: MDP $M = (S, A, T, R, s_0)$, threshold function $f$, time horizon $h$
 2: $s'_0 \leftarrow (s_0, h, 0)$
 3: $S' \leftarrow \{s'_0\}$
 4: for $i \leftarrow h$ to 1 do
 5:     for all states $s'_1 = (s_1, t, ir) \in S'$ such that $t = i$ do
 6:         for all transitions $T(s_1, a, s_2)$ in $M$ do
 7:             $s'_2 \leftarrow (s_2, t-1, ir + R(s_2))$
 8:             $S' \leftarrow S' \cup \{s'_2\}$
 9:             $T'(s'_1, a, s'_2) = T(s_1, a, s_2)$
10: for all states $s' = (s, t, ir)$ in $M'$ do
11:     if $t = 0$ then
12:         $R'(s') \leftarrow f(ir)$
13:     else
14:         $R'(s') \leftarrow 0$
15: return $M' = (S', A, T', R', s'_0)$
```

in M.[1] In the next section, we present an algorithm that solves the converted MDP M' efficiently.

Algorithm 3.2 shows our TRMDP conversion algorithm. Each state s' in the converted MDP M' is a tuple (s, t, ir), where s is a base state from M, t is the number of time steps remaining, and ir is the cumulative intermediate reward received by the agent within the first $h - t$ time steps. By design, the optimal action for a state $s' = (s, t, ir)$ in M' is the optimal action for being in some state s in M with t time steps remaining and cumulative intermediate reward of ir. Solving M' allows us to extract the optimal non-stationary policy for the TRMDP (M, f, h).

We now explain Algorithm 3.2 in detail. On line 2 of the conversion algorithm, the initial state s'_0 of M' is set to $(s_0, h, 0)$, indicating that the agent starts in state s_0, with h time steps remaining, and no cumulative intermediate reward. S' is the set of states in the converted MDP; S' initially contains only the starting state s'_0.

The first loop (lines 4–9) generates all the remaining states in S' and the new transition function T'. This loop iterates from h steps remaining down to 1; at iteration i it generates

[1]Throughout the remainder of this chapter, we will use s', $T'(s', a, s'_2)$, and $R'(s')$ to refer to the states, transitions, and rewards of the converted MDP M'. We will use s, $T(s, a, s_2)$, and $R(s)$ to refer to the base MDP M.

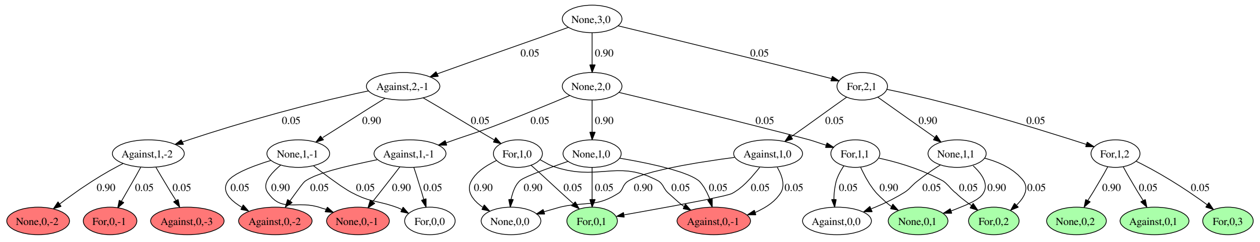


Figure 3.2: The MDP M' returned by Algorithm 3.2 given the MDP M (presented in Figure 3.1) and $h = 3$. Lightly-shaded states have reward 1; darkly-shaded states have reward -1; unshaded states have reward 0. Transition probabilities for the *balanced* action are shown.

all states that have $i - 1$ time steps remaining, by finding all the states in S' with i time steps remaining (line 5) and generating all possible successors of these states. For each such state $s'_1 = (s_1, t, ir)$, the algorithm finds all transitions in M from s_1 to some other state s_2 on action a (line 6). A new state s'_2 is created for each such s_2. Each s'_2 has base state s_2, $i - 1$ time steps remaining, and cumulative intermediate reward $ir + R(s_2)$ (line 7). s'_2 is added to S' if it doesn't already exist (line 8). The transition probability $T'(s'_1, a, s'_2)$ is equal to the transition probability $T(s_1, a, s_2)$ of the base MDP M (line 9).

The second loop (lines 10–14) assigns rewards to each state in S'. Each final state is assigned reward based on applying the threshold function f to the cumulative intermediate reward ir (line 12). Non-final states are assigned reward 0 (line 14). The algorithm has now defined S', T', R', and s'_0; the action space A is left unchanged. The converted MDP M' is then (S', A, T', R', s'_0) (line 15).

Figure 3.2 shows the result of applying the conversion algorithm to the example MDP M (with $h = 3$). Each state s' in M' is a tuple (s, t, ir), where s is a base state from M, t is the number of time steps remaining, and ir is the cumulative intermediate reward received by the agent within the first $h - t$ time steps. The agent starts in state $(\text{None}, 3, 0)$, indicating that the agent is in base state None with 3 time steps remaining and no cumulative intermediate reward. At every time step, t decreases by 1; ir increases by 1 when our team scores a goal (when we enter the For state) and decreases by 1 when our opponent scores a goal (when we enter the Against state). The final states are the only states with any reward; our agent receives reward $+1$ for a win, -1 for a loss, and 0 for a tie.

3.4 TRMDP Solution Algorithm

The optimal policy for a TRMDP (M, f, h) is the solution to M', which is generated by Algorithm 3.2 and can be solved using any MDP solution technique. The following facts about M' allow for an efficient value iteration algorithm:

Fact 1. *M' has a layered, feed-forward structure: every layer contains transitions only into the next layer.*

All MDPs generated by the conversion algorithm will have this structure due to the fact that t must decrease by 1 at every time step. (Figure 3.2 shows this fact visually.)

Fact 2. *At iteration k of value iteration, the only values that change are those for the states $s' = (s, t, ir)$ such that $t = k$.*

By design, non-zero rewards are found only in the bottom layer of the MDP; with each iteration of value iteration, these rewards propagate up to all the states in the next-higher layer. The value iteration algorithm completes after computing V^h; that is, when all the rewards have percolated up to the initial state.

These facts allow for an efficient implementation of value iteration on M': we start at the $t = 0$ layer and apply Bellman backups until the rewards "bubble up" to the top. At iteration k, we only need to calculate the value of the states in layer k of M' (that is, the states where $t = k$). Also, we do not need to sum over all states in S' when computing each value but only its $O(|S|)$ potential successors in the next lower layer (those states where $t = k - 1$.) Since each state is backed up only once, the running time of value iteration is proportional to $|S'|$, the number of states in M'.

The states of M' are arranged into $h+1$ layers. At most, each layer k will have one state for every combination of s and ir that is possible to achieve after $h - k$ steps. At the top level, there is only one possible intermediate reward value. If we assume that intermediate rewards are drawn from a set N of small integers (which is typically the case for timed, zero-sum games), then the number of possible intermediate-reward values at each subsequent layer grows by (at most) the magnitude m of the largest element in N. The number of states in layer k is therefore upper bounded by $|S| \times (h - k) \times m$, and the total number of states in the $h+1$ layers of M' is $O(|S|h^2m)$. Each Bellman update requires a maximization over $|A|$ actions of a sum of $\leq |S|$ possible successor states. Since each state is updated exactly once, the worst-case running time of value iteration on M' is $O(|A||S|^2h^2m)$.

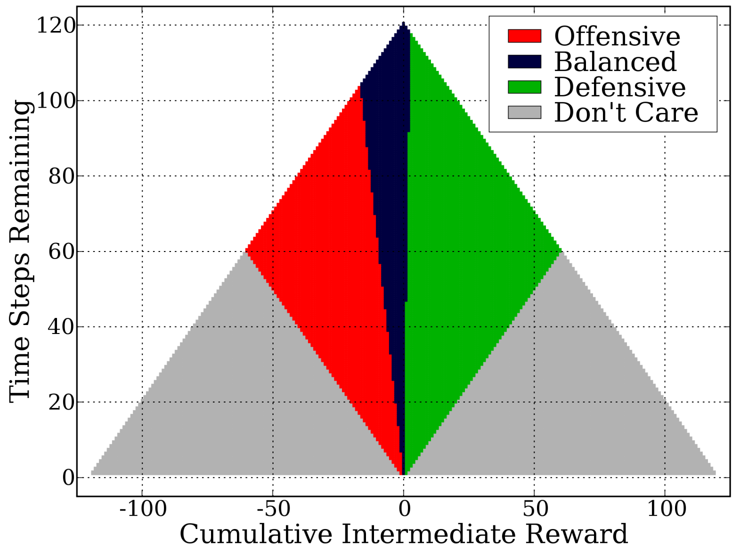


Figure 3.3: The optimal policy for M (shown in Figure 3.1), with time horizon $h = 120$ steps.

3.5 Results

Figure 3.3 shows the optimal policy for the example MDP M (Figure 3.1) with time horizon $h = 120$. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference). The shaded areas show the optimal action for every possible combination of time remaining and intermediate reward. (Since M's transition probabilities are the same from every state, the policy does not depend on the current state.) This policy has an expected reward of 0.1457. By following this policy, our agent will win approximately 50% of the time, lose 35% of the time, and tie 15% of the time.

Figure 3.3 shows that the optimal policy for M is nonstationary: the policy depends on the number of time steps remaining and the cumulative intermediate reward. Qualitatively, the optimal policy is to choose the *defensive* play when winning by a significant number of points and to choose the *offensive* play when losing by a significant number of points. When the score is close to a tie, the best play is *balanced*. As the time remaining decreases, the point difference needed to choose *offensive* or *defensive* decreases—the agent acts more urgently. The *don't care* regions in the lower-left and lower-right of the figure are states from which the actions of the agent no longer have any effect on the final outcome, because the number of steps remaining is greater than the score difference. In these regions, all actions have equal expected reward and the agent can choose between them arbitrarily.

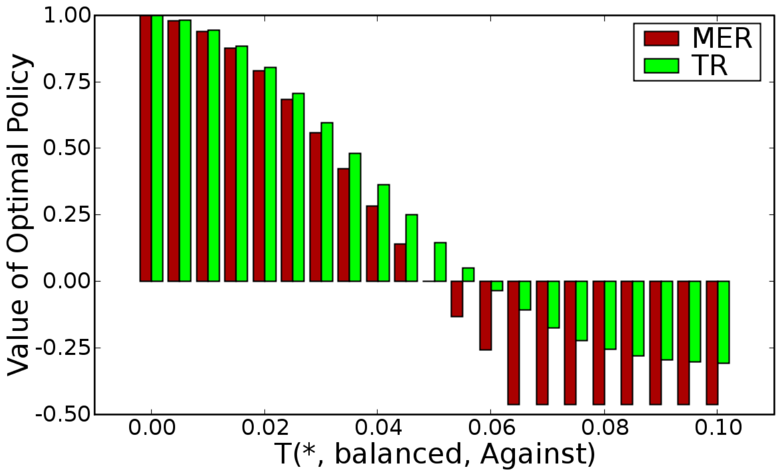


Figure 3.4: Effect of changing the opponent's capabilities.

In Figure 3.4, we show the effect of changing the opponent's capabilities. Specifically, we vary the probability $T(*,\ \textit{balanced},\ \textsc{Against})$ of our opponent scoring when we choose the *balanced* action. The y-axis shows the expected true reward of following the optimal maximize-expected-rewards policy (MER) and of following the optimal thresholded-rewards policy (TR). In all cases, TR performs better than MER. It is interesting to note that the difference between the two objective functions is greatest when the capabilities of each team are similar—that is, where $T(*,\ \textit{balanced},\ \textsc{Against})$ is close to 0.5.

We also consider the performance of MER and TR on 5000 randomly generated MDPs. Each of these MDPs has the same structure as M (shown in Figure 3.1), but the transition probabilities of each state/action pair are chosen as follows: $T(s, a,\ \textsc{Against})$ is chosen uniformly from $[0.0, 0.5)$; $T(s, a,\ \textsc{For})$ is chosen uniformly from $[0.9, 1.0) \times T(s, a,\ \textsc{Against})$; and $T(s, a,\ \textsc{None})$ is set such that the three probabilities sum to 1. Note that our team is less likely to score than the opponents at every timestep, no matter which action is chosen. Therefore, the expected true reward of MER is negative for every MDP. Figure 3.5 is a histogram depicting the distribution of true rewards for MER and TR on these 5000 MDPs. Each bar shows the number of MDPs that have optimal policies within a given range of true rewards. The mean true reward for MER is -0.0659; the mean true reward for TR is 0.1971. These results show that explicitly reasoning about score and time remaining allows our team to win with high probability, even against an opponent that is otherwise superior.

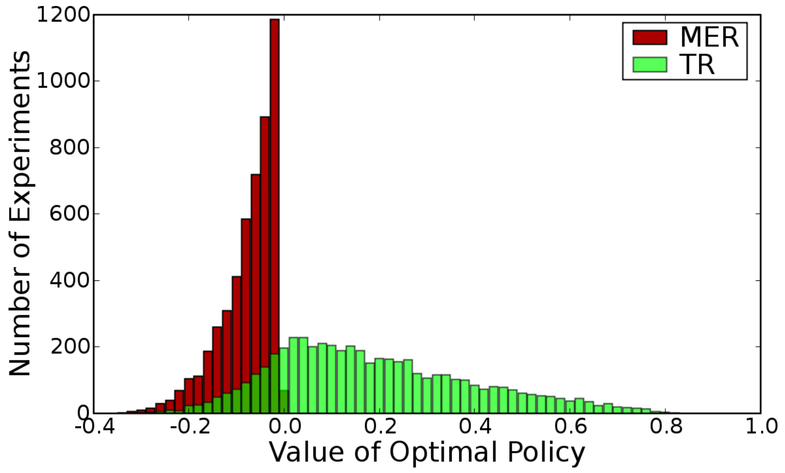


Figure 3.5: Performance of maximize-expected-rewards and thresholded-rewards on 5000 randomly generated MDPs.

3.6 Summary

In this chapter, we introduced *thresholded-rewards* problems, in which an agent gains *intermediate rewards* during execution in a finite-horizon environment. At the end of the horizon, the agent receives a *true reward*, which is determined by applying a threshold function to the intermediate rewards. Thresholded rewards are particularly applicable to timed, zero-sum games, such as robot soccer and Capture the Flag. In these domains, thresholded rewards allow us to maximize the probability of winning. Thresholded rewards are also applicable in other stochastic domains with a hard deadline, such as reCAPTCHA; in these domains, thresholded rewards can be used to maximize the probability of achieving a given amount of reward before the deadline.

We have presented an algorithm that converts a base MDP into an expanded MDP suitable for solving thresholded-rewards problems. Solving this expanded MDP yields the optimal, nonstationary policy for the original MDP. In this chapter, we focus on an efficient value iteration algorithm which finds solutions to thresholded-rewards MDPs in $O(|A||S|^2h^2m)$ time.

As presented in this chapter, thresholded-rewards MDPs assume that the opponent is static and can be treated as part of the environment. If the opponent is unknown, or changes its strategy during the course of the game, we cannot model the opponent as a static part of the environment. In Chapter 6, we consider relaxing these assumptions. In the next chapter, we present a variety of heuristic solution techniques that can be used to efficiently find approximately optimal policies for thresholded-rewards MDPs.

Chapter 4

Heuristics for TRMDPs

In Chapter 3, we introduced an algorithm that converts a base MDP into an expanded MDP suitable for solving thresholded-rewards problems. This algorithm finds solutions to thresholded-rewards MDPs in $O(|A||S|^2h^2m)$ time. This quadratic dependence on state space size and time horizon length means that the exact algorithm may be computationally intractable for MDPs with many thousands of states or with time horizons that are several thousand time steps long. In this chapter, we introduce three different heuristic techniques that find approximate solutions to TRMDPs. These heuristics trade off computation time with solution quality, achieving performance close to the optimal solution while using significantly less computation time. Each of these heuristics can be seen as an informed version of state aggregation [28], in which states are aggregated based on the time remaining. This state aggregation is in essence equivalent to changing the resolution of time discretization.

4.1 The *Uniform-k* Heuristic

With the *uniform-k* heuristic, our agent adopts a nonstationary policy (mapping from states in the base MDP to actions) but only considers changing its policy every k time steps. The net effect of this change is to "compress" the time horizon uniformly by a factor of k, directly leading to a decrease in the state space of the expanded MDP M'. This solution is more efficient, but suboptimal because it does not consider switching policies at every time step.

Figure 4.1 shows the *uniform*-10 policy for our example domain with time horizon $h = 120$ steps. Only the policy decision times are shown; at all other time steps, the last decision is

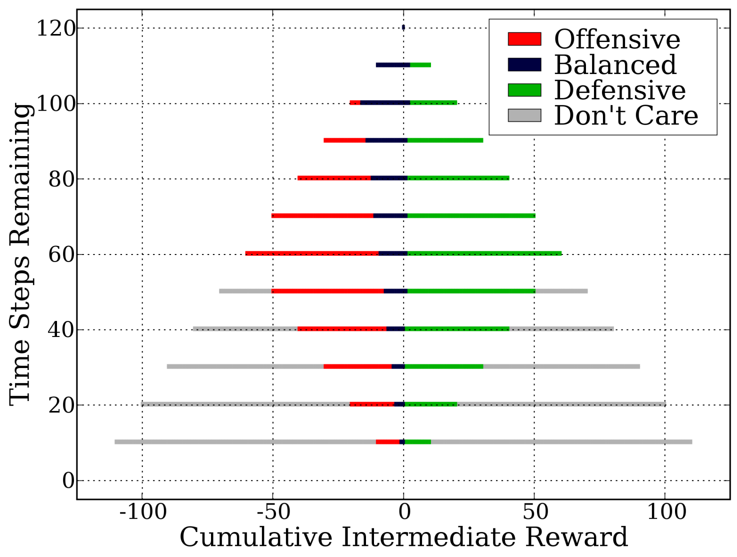


Figure 4.1: The *uniform*-10 policy for M with time horizon $h = 120$ steps. Only the policy decision times are shown; at all other time steps, the last decision is maintained.

maintained. The *uniform*-10 policy has the same general form as the optimal policy, but since it only needs to make a decision every 10 steps, the number of states in the expanded MDP (and therefore the time required to compute the optimal solution) is significantly reduced.

To compute the *uniform* policy, we need to change the TRMDP conversion algorithm (Algorithm 3.2) slightly, specifically the computation of the expanded states (lines 6–7). The optimal algorithm uses the one-step transition function of the base MDP to compute new expanded states. With *uniform-k*, we instead need a k-step transition function. Formally, let $T_k(s_1, a, s_2)$ be the function that returns the probability that we end up in state s_2, given that we start in state s_1 and choose action a for all of the next k time steps. T_k can be defined recursively, as follows:

Base case: $T_1(s_1, a, s_2) = T(s_1, a, s_2)$.
The one-step transition probability is calculated directly from the transition function of the base MDP.

Inductive case: $T_k(s_1, a, s_2) = \sum_{s_3 \in S}(T_{k-1}(s_1, a, s_3) \times T_1(s_3, a, s_2))$.
We sum over all states s_3 that are reachable in $k-1$ time steps. For every such state s_3, we find the probability of ending up in s_3 if we start in s_1 and choose action a for $k-1$ steps, and multiply this value by the one-step transition probability from s_3 to s_2 on action a.

A simple dynamic programming algorithm uses these recurrences to efficiently compute the k-step transition function $T_k(s_1, a, s_2)$ for every possible combination of s_1, a, and s_2. T_k is precomputed before running the TRMDP conversion algorithm; line 6 of Algorithm 3.2 then uses T_k instead of T. One related change is needed: on line 7 of Algorithm 3.2, we need to consider the distribution of reward values accumulated during the k steps from s_1 to s_2 instead of simply using $R(s_2)$. However, this distribution of reward values can be precomputed alongside T_k, using an analogous recurrence.

The resulting expanded MDP has fewer states than the optimal MDP. The h levels of the optimal MDP are replaced by h/k levels in the uniform-k approximation; the resulting number of states in the uniform-k MDP is $O(|S|\frac{h^2}{k}m)$, which is smaller by a factor of k. For most MDPs, this smaller state space size directly corresponds to a decrease in the time needed to solve the expanded MDP. However, in the worst case, the average state in the uniform-k MDP might have more successors than the average state in the optimal MDP (since we are taking k steps "at once," there are more possible final states). If the average number of transitions per state in the uniform-k MDP is a factor of k bigger, we have not gained anything: the Bellman updates for each state will take k times as long, and the overall solution time will remain unchanged. However, for most domains, the average number of transitions per state in the uniform-k MDP will not increase significantly, so the computation of the uniform-k policy will directly correspond to a decrease in the time needed to solve the expanded MDP.

In addition to its use as a heuristic solution strategy, *uniform-k* also lets us measure how the discretization of time affects performance. The difference between the values of the optimal and *uniform-k* policies tells us how much would be lost if each time step lasted k times as long.

4.2 The *Lazy-k* Heuristic

With the *lazy-k* heuristic, our agent ignores time and score until there are k steps remaining. For the first $h-k$ time steps, the *lazy-k* policy acts in accordance with the optimal maximize-expected-rewards (not thresholded-rewards) policy for the base MDP M. Once there are k steps remaining, the agent uses Algorithm 3.2 to create an optimal TRMDP M' with a time horizon k and an initial state chosen to reflect the actual current state of the system (including the cumulative intermediate reward). The agent solves M' and uses the optimal thresholded-rewards solution to adopt a nonstationary policy for the remaining k time steps. The main idea of this technique is to concentrate computational effort near the end of the run, when the agent's actions may have a greater effect on the overall outcome.

Figure 4.2 shows the *lazy*-30 policy for our example domain. For the first 90 time steps (at the top of the figure), the agent acts optimally with respect to the base MDP, always choosing the *balanced* action. For the remaining 30 time steps, the agent acts optimally with respect to the expanded TRMDP. The figure shows the case where our agent happens to be ahead by 3 points when there are 30 time steps left.

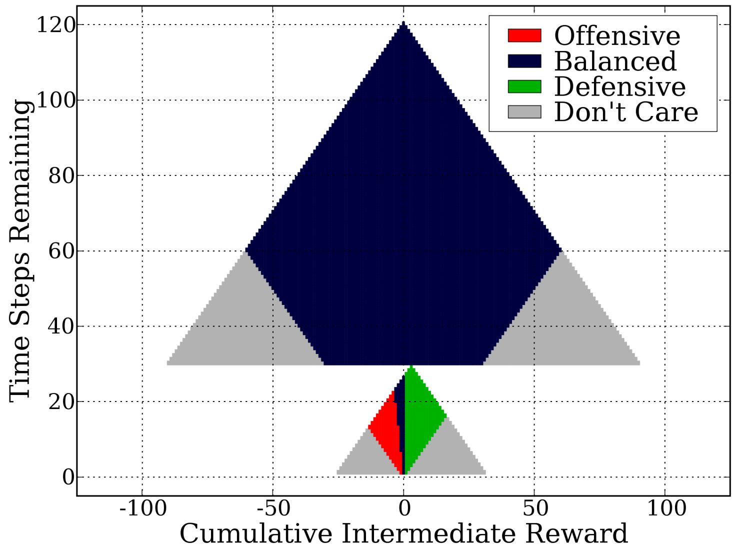


Figure 4.2: The *lazy*-30 policy for M with time horizon $h = 120$ steps.

4.3 The *Logarithmic-k-m* Heuristic

With this heuristic, the agent makes a number of decisions that is logarithmic in the time horizon. The *lazy-k* heuristic saves computation by only considering the end of the time horizon; the *uniform-k* heuristic saves computation by compressing the entire time horizon. The *logarithmic* heuristic is a hybrid approach in which the time resolution becomes finer as we approach the end of the time horizon.

For example, the agent may switch policies at every step during the final 8 steps of the run, every two steps for the previous 16 steps of the run, every four steps for the previous 32 steps of the run, and so on. The *logarithmic* heuristic depends on two parameters: k, the number of decisions the agent makes before the time resolution is increased, and m, the multiple by which the resolution is increased. For this example, $k = 8$ because the agent takes 8 actions before each increase, and $m = 2$ because the time resolution doubles on each

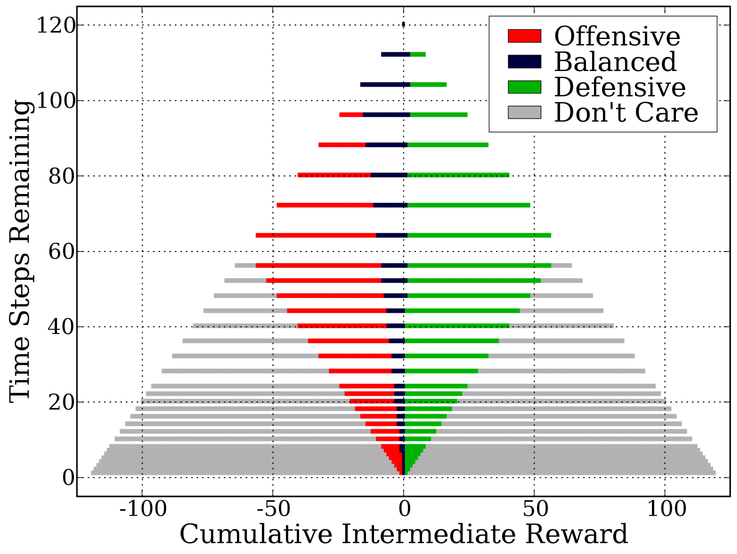


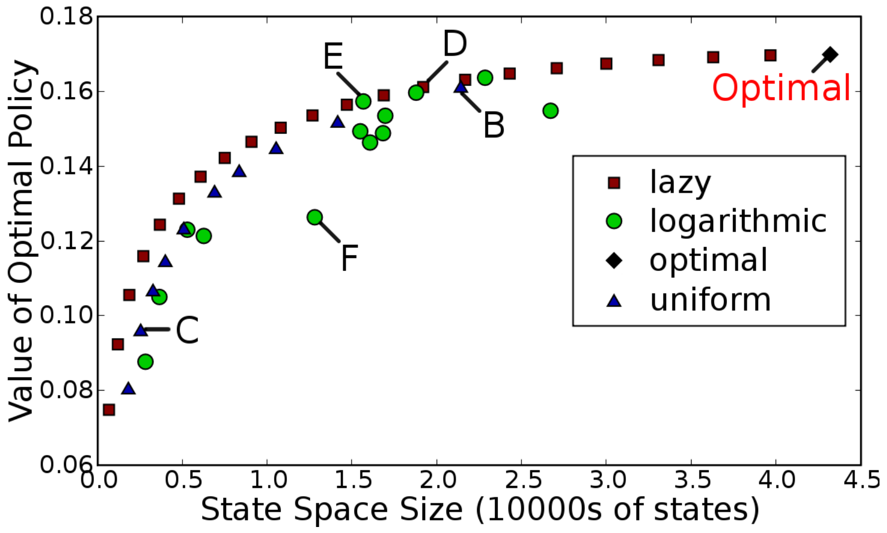
Figure 4.3: The *logarithmic*-8-2 policy for M with time horizon $h = 120$ steps. Only the policy decision times are shown; at all other time steps, the last decision is maintained.

increase. Figure 4.3 shows the *logarithmic*-8-2 policy for our example domain. Like *uniform*, the *logarithmic* policy has the same general form as the optimal TRMDP policy, but the agent allocates most of its computational time (MDP states) on accurately modeling the domain near the end of the time horizon. At the end of the game, the agent makes action choices at every time step, but for the first half of the game the agent makes action choices only every 8 steps. In between, there is a period of 4-step gaps and then a period of 2-step gaps.

The *logarithmic* heuristic also requires T_n, the n-step transition and reward functions, which can be computed as in the *uniform* heuristic. The difference between *logarithmic* and *uniform* is on line 6 of Algorithm 3.2: rather than using a fixed n in T_n, *logarithmic* varies n depending on how many time steps are remaining and the values of k and m.

4.4 Results

We tested the performance of these three heuristic techniques, for a variety of parameter settings, on 60 different MDPs. These MDPs were chosen randomly from the 5000 MDPs that were used in Section 3.4. Figure 4.4 summarizes the results. Each point on the graph corresponds to a heuristic technique with some parameter setting. The x-axis shows the number of states in the expanded MDP (averaged over the 60 MDPs); the y-axis shows the mean expected true reward of that heuristic technique. Ideally, we would like a technique that provides a high true reward with a low number of states. Points in the upper-left frontier of the graph represent Pareto-efficient tradeoffs between state space size and expected true reward.



Point	Approach	Value	# States
Optimal	Optimal	0.1699	43,200
B	*Uniform*-2	0.1608	21,240
C	*Uniform*-15	0.0957	2,544
D	*Lazy*-80	0.1612	19,200
E	*Logarithmic*-8-2	0.1573	15,672
F	*Logarithmic*-2-4	0.1264	12,807

Figure 4.4: Performance of heuristic techniques on 60 randomly generated MDPs. Points in the upper-left frontier of the graph represent Pareto-efficient tradeoffs between state space size and expected true reward.

The optimal thresholded-rewards algorithm, labeled "Optimal" in the graph, has a mean reward of 0.1699 and requires 43,200 states to compute. For small values of k, the *uniform* heuristic closely approximates the optimal solution while significantly reducing the size of the state space. *Uniform*-2, labeled "B", has mean reward 0.1608 and requires 21,420 states. The selection of k is a tradeoff between solution time and quality; *uniform*-15 (labeled "C") uses only 2,544 states, but the reward drops to 0.0957. *Lazy*-80 (labeled "D") has mean reward 0.1612 and uses only 19,200 states; this is fewer states than *uniform*-2 and a higher mean reward. In general, the *lazy* heuristic consistently has a higher reward than *uniform* at a given state space size. *Logarithmic*-8-2, labeled "E", closely matches the performance of *lazy-k*; however, *logarithmic*-2-4 (labeled "F") performs much worse than both *lazy* and *uniform*. In general, the performance of *logarithmic* seems highly parameter-dependent, and in no case does *logarithmic* significantly outperform *lazy* at a given state space size.

Of the heuristics introduced in this chapter, *lazy* consistently offers the best tradeoff in terms of solution time and quality, which seems to indicate that acting optimally is most important near the end of the time horizon. In fact, we can provide some support for this intuition. Specifically, consider a policy called "Optimal-except-k". With this policy, our agent acts optimally (according to the optimal thresholded-rewards policy), except at a single time step, k. At time step k, the agent instead acts according to the optimal maximize-expected-rewards action (*balanced*). We evaluate this policy and compute the loss (decrease in expected value) compared to optimal. Figure 4.5 shows the loss for each possible value of k. This figure shows that acting suboptimally near the end of the time horizon (when k is close to 0) causes a much greater loss than acting suboptimally near the beginning of the time horizon (when k is close to 120.)

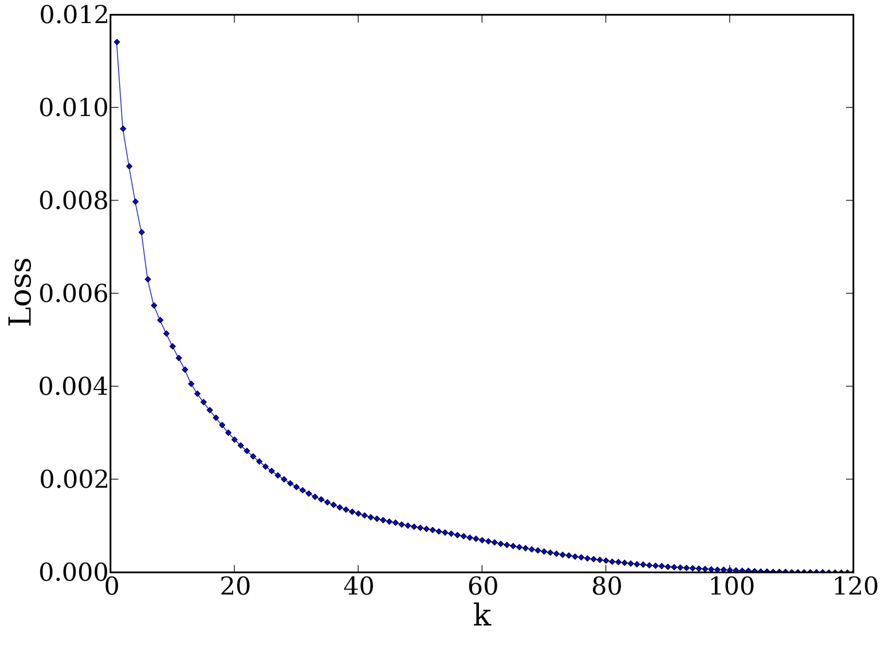


Figure 4.5: The amount of expected value lost if an agent acts optimally except at a single time step k, where $k = 0$ is the final time step.

4.5 Summary

In this chapter, we introduced three heuristic techniques that find approximately optimal policies for thresholded-rewards MDPs. These heuristics trade off computation time with solution quality, achieving performance close to the optimal solution while using significantly less computation time. With the *uniform-k* heuristic, our agent adopts a nonstationary policy but only considers changing its policy every k time steps. This heuristic "compresses" the time horizon uniformly by a factor of k, directly leading to a decrease in the state space of the expanded MDP. With the *lazy-k* heuristic, our agent ignores time and score until there are k steps remaining. The *lazy-k* heuristic concentrates computational effort near the end of the run, when the agent's actions may have a greater effect on the overall outcome. With the *logarithmic-k-m* heuristic, the agent makes a number of decisions that is logarithmic in the time horizon. The *logarithmic-k-m* heuristic is a hybrid approach in which the time resolution becomes finer as we approach the end of the time horizon. Of these three heuristics, the *lazy-k* heuristic consistently has the highest performance at a given state space size.

Chapter 5

TRMDPs with Arbitrary Reward Distributions

In Chapter 3, we introduced TRMDPs, which find a policy that maximizes the probability of achieving a specified amount of reward in timed domains. With the TRMDP model applied to the robot soccer domain, each state-action pair has a fixed probability that a goal will be scored by one team or the other. This implies that, as long as the system remains in a given state, the amount of time taken to score is drawn from a geometric distribution. For many games, including robot soccer and CTF, this assumption is inaccurate. For example, even if the ball is very close to the goal in robot soccer, it may take a significant amount of time for the attacking robot to score or for the defending goalkeeper to clear the ball to safe territory. We therefore propose that, in rich domains, it is more accurate to include semi-Markov actions, which are temporally extended and may take multiple time steps to complete. In this chapter, we augment TRMDPs by adding semi-Markov actions, thereby introducing thresholded-rewards semi-Markov Decision Processes (TRSMDPs). TRSMDPs model thresholded-rewards domains in which the time needed for a state transition or reward assignment follows any arbitrary distribution.

Section 5.1 presents our formal definition of TRSMDPs. Section 5.2 presents an optimal solution algorithm for TRSMDPs. Section 5.3 presents our CTF results, including the time-to-score distributions for each combination of CTF plays, the optimal TRSMDP policies against each opponent play in the CTF domain, and empirical evaluations of these policies. Section 5.4 presents results with our real robot soccer team, including time-to-score distributions of two different plays and the optimal TRSMDP policy against a static opponent. In Section 5.5, we use an MDP-based approach to derive optimal policies for the CTF domain, and empirically compare the effectiveness of the resulting policies with the SMDP-based approach. Section 5.6 introduces the *threshold-plus-linear-k* objective function, which trades off between maximizing the probability of winning and maximizing the margin of victory.

5.1 Definitions

Similarly to Chapter 3, we represent an SMDP as a tuple (S, A, T, s_0). The transition function T is augmented to describe the probability distribution over the next state, the amount of reward achieved, *and* the amount of time it takes for the transition to occur. In the robot soccer and CTF domains, we measure the transition functions empirically, building a model of each domain's transition dynamics offline. During this offline phase, our team repeatedly chooses an action from some given state; after each action, we observe the next state, the amount of time elapsed, and the reward received. Formally, given a state/action pair (s, a) as input, the world produces an *outcome* (s', dt, r). Given many such outcomes for each possible state/action pair, we generate an estimated transition function $\hat{T}$. $\hat{T}(s, a)$ returns a list of tuples of the form (p, s', dt, r), where p is the probability of transitioning to state s' after dt time steps, receiving reward r, when action a is executed in state s. A *thresholded-rewards SMDP* (TRSMDP) is defined by the tuple (M, f, h), where M is an SMDP, f is the reward threshold function, and h is the time horizon.

5.2 TRSMDP Optimal Solution Algorithm

Algorithm 5.1 is a function that computes the value of a given state (s, t, ir) of a TRSMDP via a straightforward recursive computation of the standard Bellman equation [69]. In addition, this function also returns the optimal action selection for this state. Algorithm 5.1 takes in a state $s \in S$, the time remaining t, the cumulative intermediate reward ir, the set of actions A, the transition function T, and the reward threshold function f.

If there is no time left, the value of this state is the result of calling the threshold function with the amount of cumulative intermediate reward (lines 2–3). There is no "best action" to return since it is not possible to take another action when there is no time remaining.

Otherwise, the algorithm computes the value of each action and finds which action has the greatest value (lines 4–18). To compute the value of an action, the algorithm finds all possible transitions (line 8) and sums up the values of the resulting states, multiplied by the probability of that transition occuring (lines 8–15). On line 9, the algorithm computes

Algorithm 5.1 Computes the value of a given state of a TRSMDP.

```
 1: function StateValue($s, t, ir, A, T, f$):
 2: if $t = 0$ then
 3:     return ($f(ir)$, None)
 4: best_action ← None
 5: best_action_value ← $-\infty$
 6: for $a \in A$ do
 7:     action_value = 0
 8:     for $(p, s', dt, r) \in T(s, a)$ do
 9:         $t' \leftarrow t - dt$
10:         if $t' < 0$ then
11:             $t' \leftarrow 0$
12:             $ir' \leftarrow ir$
13:         else
14:             $ir' \leftarrow ir + r$
15:         action_value ← action_value + $p \times$ StateValue($s', t', ir', A, T, f$)
16:     if action_value > best_action_value then
17:         best_action ← $a$
18:         best_action_value ← action_value
19: return (best_action_value, best_action)
```

the new time remaining t' by subtracting the transition time dt from the previous time remaining t. It might be the case that dt, the amount of time needed for the transition to occur, is greater than t, the amount of time remaining before the time horizon. In this case, $t' < 0$, and the action does not actually complete before the time horizon ends. Therefore no additional intermediate reward is received. Lines 10–12 handle the case where $t' < 0$. Otherwise, the algorithm adds the reward to our cumulative intermediate reward as usual (lines 13–14). Line 15 is the recursive step; the action's value is increased by the probability p of the transition occurring multiplied by the value of the new state (s', t', ir'). If a's value is higher than any other action seen so far, we set a to be the best action and store its value as the best action value seen so far (lines 16–18). After iterating over all actions, we return the best action and its value (line 19).

As presented, Algorithm 5.1 is computationally inefficient: the recursive call to StateValue on line 15 gets called with the same exact arguments repeatedly, because each state (s', t', ir') is potentially the successor state of many other states. Each repeated call to StateValue with the same arguments recomputes the results each time. A significant performance gain is achieved by using the well-known technique of *memoization*. We simply keep a cache that maps (s, t, ir, A, T, f) tuples to (best_value, best_action) tuples. If StateValue is called with an (s, t, ir, A, T, f) tuple that is already in the cache, StateValue returns the precomputed value and optimal action immediately; otherwise StateValue computes the value using the algorithm given above and stores the result in the cache for future use. For all results presented in this thesis, we used memoization; the number of entries needed in the cache was small enough that the entire cache easily fit in memory (taking up less than 80MB of RAM in all cases).

Algorithm 5.1 only computes the value and optimal action for a single state; by itself, it does not provide a complete policy for a TRSMDP (M, f, h). However, the memoized version does compute the entire policy. Assuming that the transition function T is exact, the memoized computation of StateValue$(s_0, h, 0, A, T, f)$ recursively computes the value of every reachable state (s, t, ir), storing the value and the optimal action of each reachable state in the cache. The cache can then be saved to disk so that the optimal policy is available at runtime.

However, in both the CTF and robot soccer domains, we use an estimate $\hat{T}$ of the transition function which is found empirically, by playing multiple games and keeping track of all the transitions seen. Therefore, at runtime the agent could potentially end up in a state (s, t, ir) which is "impossible" given the estimated transition function $\hat{T}$ but which is in fact possible with the domain's true transition function T (which is unknown). Assuming that the estimated transition function is fairly accurate, there should be relatively few "impossible" states. However, to ensure that *every* state reachable at runtime has an optimal action computed, we just need to make sure to call StateValue(s, t, ir, A, T, f) for every value of s, t, and ir. Algorithm 5.2 achieves this with a simple nested loop over all possible (s, t, ir) values. If there are relatively few "impossible" states, this entire loop will not take significantly more computation time than simply computing the value of the initial state $(s_0, h, 0)$. For example, in the CTF experiments presented in the following section, computing the entire policy takes approximately 3.8 times longer than computing the value of the initial state.

Algorithm 5.2 requires the enumeration of all possible intermediate reward values. As in Chapter 3, we could assume that the reward at each time step grows by the maximum-magnitude single-step intermediate reward. Then, the possible intermediate reward values in a 2000-step CTF game are bounded by $[-2000, 2000]$. However, SMDPs often allow a stronger assumption that limits the number of possible intermediate reward values further. For example, in the CTF domain with a 72×48 world, at least 53 steps are required in order to score a point: the flag is initially 26 squares away from midfield, so in order to score, a player must make at least 26 `MoveAction`s toward the flag, one `PickupAction` to pick up the flag, and 26 more `MoveAction`s to successfully carry it back to the home zone. Therefore, in a CTF game of 2000 time steps, the maximum number of scores is upper bounded by

Algorithm 5.2 Computes the optimal policy for every state of a TRSMDP.

```
1: Given: SMDP $M = (S, A, T, s_0)$, threshold function $f$, time horizon $h$:
2: for $s \in S$ do
3:     for $t \in \{0, 1, \ldots h\}$ do
4:         for $ir \in$ { all possible values of cumulative intermediate reward } do
5:             $\pi(s, t, ir) \leftarrow V(s, t, ir, A, T, f)$
6: return $\pi$
```

$\lfloor 2000/53 \rfloor = 37$, which gives a much tighter bound on the possible values of ir.

5.3 TRSMDPs Applied to the CTF Domain

In this section, we apply the TRSMDP solution algorithm to find optimal policies for the CTF domain. Section 2.2 presents full details of the CTF domain. In order to find an optimal policy for the CTF domain, we need to model the domain as an SMDP (S, A, T, s_0). Since we are primarily interested in modeling strategic decisions and the effects of team actions, our set of actions A is the 11 team-level CTF plays introduced in Section 2.2.3. We analyze these plays in Section 5.3.1. To highlight the effects of team strategies further, we choose to represent the CTF domain as containing only a single state. Our main concern is then the transitions: given our choice of play and the opponent's choice of play, what is the distribution of scores? Since CTF is a two-player game, the transition function $T(s, a)$ takes as input the *joint action* $a = (a^b, a^r)$ chosen by the blue and red teams, respectively. Since CTF is a timed, zero-sum domain, the evaluation of each play combination depends on the number of points scored by each team, and in the distribution of the amount of time it takes to achieve a score. Section 5.3.2 discusses these time-to-score distributions in detail. Section 5.3.3 ties these results together and presents the optimal TRSMDP policies for the CTF domain. There is one optimal policy in response to each of the possible opponent plays. Finally, Section 5.3.4 tests these optimal policies empirically, by playing each optimal policy against its associated opponent play.

Throughout this chapter, we assume that the opponent is playing a single play throughout the entire game, and that we have perfect knowledge of which play the opponent is playing. (We relax these assumptions in the next chapter.)

5.3.1 Finding Good CTF Plays

In Section 2.2.3, we introduced the eleven plays which are used in all of our CTF experiments. To judge the usefulness of these eleven plays, we perform a preliminary test in which each play is played against each other play, for a total of 121 play combinations. Each play combination was tested out for 500 games; each game is 2000 time steps long. For each combination, we recorded the number of wins achieved by each team and the total cumulative score achieved by each team over the 500 games. The full results of these games are presented in Section B.1 in Appendix B; in this section, we highlight a selection of the full results.

We apply *iterated dominance* [18] to eliminate plays which are strictly worse against all of the other possible plays, because it is unlikely that a team would ever choose to utilize such a play. Formally, we say that a play p_1 is *Pareto dominated* by play p_2 if, for every possible opponent play p_3:

- Our team scores more points when playing p_2 rather than p_1, **and**
- The opponent scores fewer points when our team plays p_2 rather than p_1.

Any play which is Pareto dominated by some other play can be removed from further consideration, since there is no situation in which a rational team would use that play.

As an example, we compare the performance of the plays `A0_M0_D5` and `A0_M1_D4`, which can be qualitatively considered the "most defensive" plays since neither play assigns any attackers. Table 5.1 shows the number of points scored by the opposing team for each play combination. Note that, for every opponent play, the opponent scores more points against `A0_M0_D5` than against `A0_M1_D4`. Since neither play assigns attackers, our team never scores any points with either play. Therefore, `A0_M0_D5` is Pareto dominated by `A0_M1_D4`. There is no situation in which a rational team would ever choose to play `A0_M0_D5`, so we remove `A0_M0_D5` from consideration.

As a further example, Table 5.2 compares `A4_M0_D1` and `A4_M1_D0` in a similar fashion. The table shows that, for nearly every opponent play, our team scores more points, and gives up fewer opponent points, if our team plays `A4_M1_D0` rather than `A4_M0_D1`. The only opponent play for which this is not the case is `A0_M0_D5`; however, we have already shown that there is no situation in which a rational opponent would choose to play `A0_M0_D5`. Therefore, there is no plausible situation in which we would choose to play `A4_M0_D1`, so we also remove `A4_M0_D1` from consideration.

If we analyze all pairs of plays in this fashion, it turns out that every play with k attackers,

Our Play	Opponent Play	Our Points	Opponent Points
A0_M0_D5	A0_M0_D5	0	0
A0_M1_D4	A0_M0_D5	0	0
A0_M0_D5	A0_M1_D4	0	0
A0_M1_D4	A0_M1_D4	0	0
A0_M0_D5	A1_M0_D4	0	104
A0_M1_D4	A1_M0_D4	0	37
A0_M0_D5	A1_M1_D3	0	65
A0_M1_D4	A1_M1_D3	0	32
A0_M0_D5	A2_M0_D3	0	171
A0_M1_D4	A2_M0_D3	0	126
A0_M0_D5	A2_M1_D2	0	557
A0_M1_D4	A2_M1_D2	0	306
A0_M0_D5	A3_M0_D2	0	814
A0_M1_D4	A3_M0_D2	0	557
A0_M0_D5	A3_M1_D1	0	774
A0_M1_D4	A3_M1_D1	0	559
A0_M0_D5	A4_M0_D1	0	996
A0_M1_D4	A4_M0_D1	0	767
A0_M0_D5	A4_M1_D0	0	982
A0_M1_D4	A4_M1_D0	0	771
A0_M0_D5	A5_M0_D0	0	1208
A0_M1_D4	A5_M0_D0	0	996

Table 5.1: Pareto-dominance analysis of two CTF plays: `A0_M0_D5` and `A0_M1_D4`.

0 midfielders, and $5 - k$ defenders is Pareto dominated by the play with k attackers, 1 midfielder, and $4 - k$ defenders. In other words, if a play assigns a defender but no midfielder, the play which replaces one defender with a midfielder is better.

The only good play containing no midfielders is `A5_M0_D0`, in which all players become attackers; this is the best play in response to a opponent play which assigns no attackers, such as `A0_M1_D4`. Of the eleven total plays, only six of them are feasible according to our Pareto-dominance analysis: `A0_M1_D4`, `A1_M1_D3`, `A2_M1_D2`, `A3_M1_D1`, `A4_M1_D0`, and `A5_M0_D0`. We consider only these six plays throughout the remainder of this document.

Our Play	Opponent Play	# Points For	# Points Against
A4_M0_D1	A0_M0_D5	1048	0
A4_M1_D0	A0_M0_D5	963	0
A4_M0_D1	A0_M1_D4	713	0
A4_M1_D0	A0_M1_D4	745	0
A4_M0_D1	A1_M0_D4	2207	2832
A4_M1_D0	A1_M0_D4	2586	846
A4_M0_D1	A1_M1_D3	359	3320
A4_M1_D0	A1_M1_D3	484	1089
A4_M0_D1	A2_M0_D3	1997	4530
A4_M1_D0	A2_M0_D3	2419	1757
A4_M0_D1	A2_M1_D2	485	5380
A4_M1_D0	A2_M1_D2	1048	2165
A4_M0_D1	A3_M0_D2	2397	5325
A4_M1_D0	A3_M0_D2	3615	1995
A4_M0_D1	A3_M1_D1	863	6340
A4_M1_D0	A3_M1_D1	1872	2433
A4_M0_D1	A4_M0_D1	4586	4499
A4_M1_D0	A4_M0_D1	6955	1055
A4_M0_D1	A4_M1_D0	1100	6902
A4_M1_D0	A4_M1_D0	2571	2594
A4_M0_D1	A5_M0_D0	6502	3442
A4_M1_D0	A5_M0_D0	8730	743

Table 5.2: Pareto-dominance analysis of two CTF plays: A4_M0_D1 and A4_M1_D0.

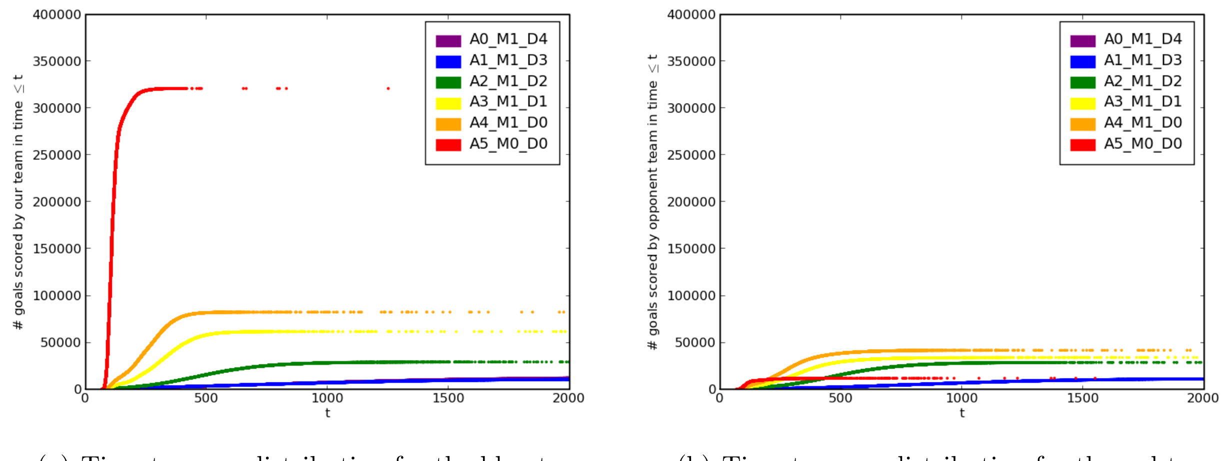
5.3.2 CTF Time-To-Score Distributions

In order to find the optimal policy for Capture the Flag, we need an accurate estimate $\hat{T}$ of the domain's transition function $T(s,a)$. $T(s,a)$ returns a list of tuples of the form (p, s', dt, r), where p is the probability of transitioning to state s' after dt time steps, with reward r, when action a is executed in state s. Since we are modeling the domain as having only a single state s_0 (see Section 5.3), we need to measure the probability distribution of r (the scores for or against our team) and dt (the amount of time each score requires) for every possible joint action $a = (a^b, a^r)$. We call these probability distributions the *time-to-score distributions*. Since each team is equipped with six different plays, there are 36 different joint actions, and 36 different time-to-score distributions that must be measured.

To measure the time-to-score distribution for a joint action $a = (a^b, a^r)$, we ran the Capture the Flag simulator for many time steps, with the blue team playing a^b and the red team playing a^r. We recorded each score as it occurred, along with the amount of time elapsed since the last score. The end result is a list L_a of n_a *outcomes*, where each outcome is a tuple (r, dt) of reward and time elapsed. We use this empirical distribution directly in our estimate of the transition function: if an outcome (r, dt) appears k times in L_a, then $\hat{T}(s_0, a)$ contains the tuple $(k/n_a, s_0, dt, r)$.

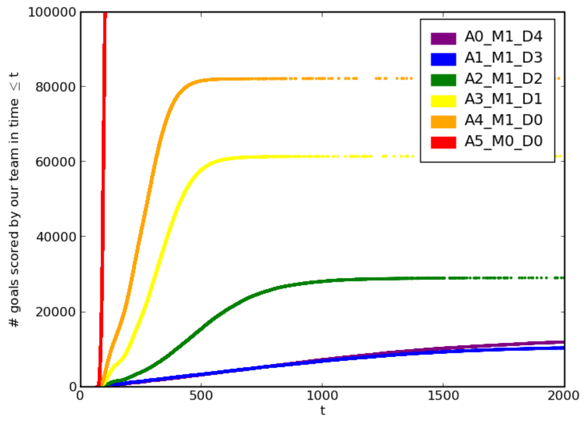
By convention, reward values of 1 and -1 correspond to scores by the blue and red team, respectively. If the simulator ran for over 2000 time steps without a point being scored, this means that this particular outcome could not lead to a score in an actual game, since games are 2000 time steps long. We therefore recorded the outcome $(0, \infty)$, indicating that no point was scored and the amount of time elapsed was effectively infinite. The simulator was then reset back to its initial state (as though a point were scored) and the simulator continued to record data.

To generate the time-to-score distributions used in the rest of this thesis, all 36 play combinations were tested for 40 million steps each. 40 million time steps is equivalent to 20000 full games of 2000 time steps; this generated enough data to ensure that the estimated transition function $\hat{T}$ very closely matches the true transition function T.

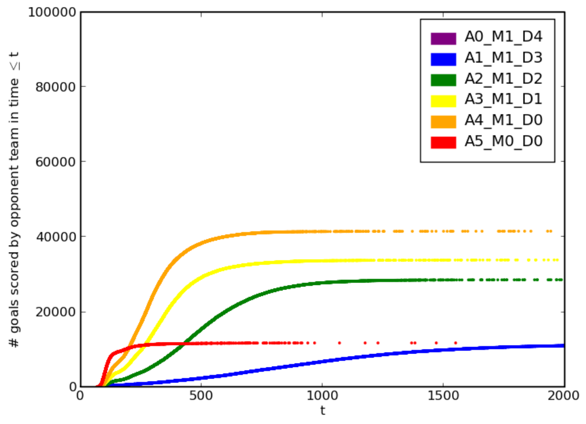


(a) Time-to-score distribution for the blue team. (b) Time-to-score distribution for the red team.

Figure 5.1: Cumulative time-to-score distributions for (a) the blue team and (b) the red team when the blue team plays `A2_M1_D2`, for each possible red play.

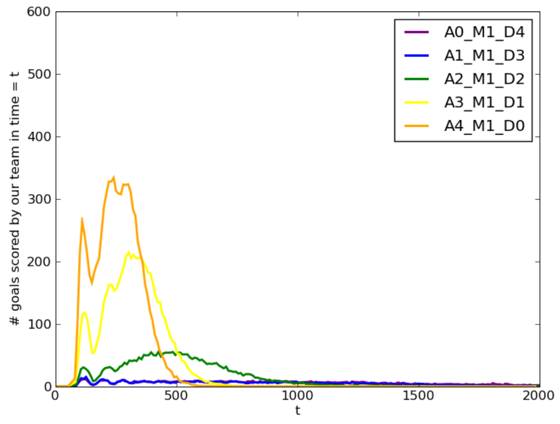


(a) Time-to-score distribution for the blue team.

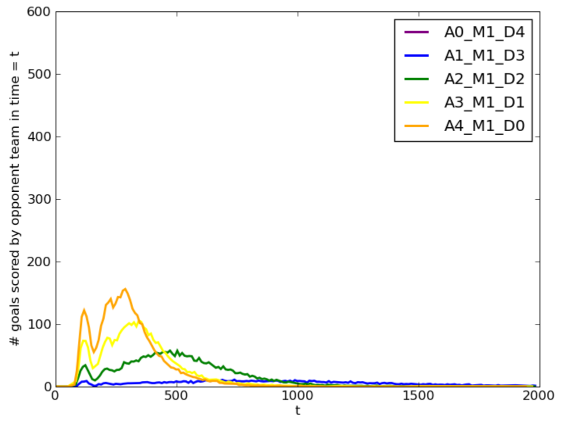


(b) Time-to-score distribution for the red team.

Figure 5.2: Cumulative time-to-score distributions for (a) the blue team and (b) the red team when the blue team plays A2_M1_D2, for each possible red play. This is the same data as Figure 5.1, with the y-axis zoomed in to highlight the differences between plays.

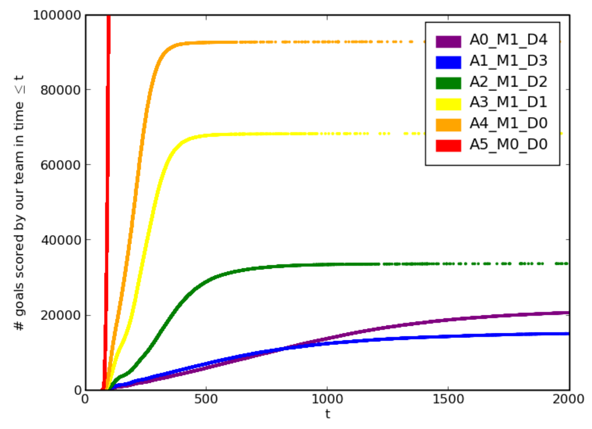


(a) Time-to-score distribution for the blue team.

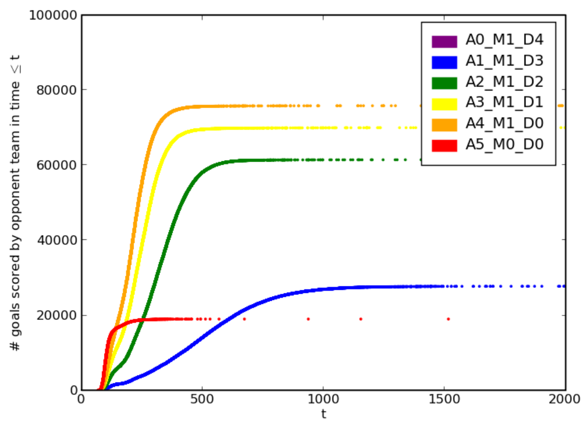


(b) Time-to-score distribution for the red team.

Figure 5.3: Instantaneous time-to-score distributions for (a) the blue team and (b) the red team when the blue team plays A2_M1_D2, for each possible red play. This is the same data as Figure 5.2, but plotted as an (unnormalized) probability mass function rather than as a cumulative distribution function.

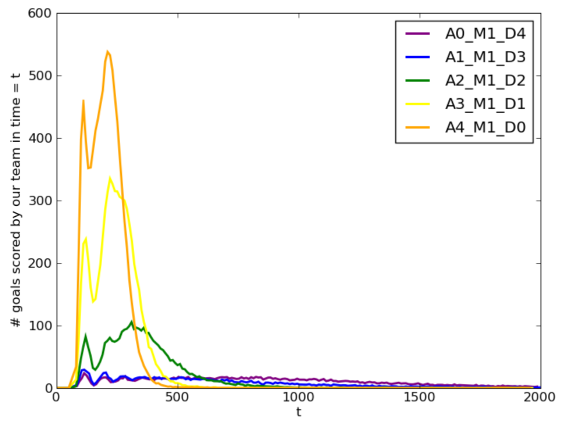


(a) Time-to-score distribution for the blue team.

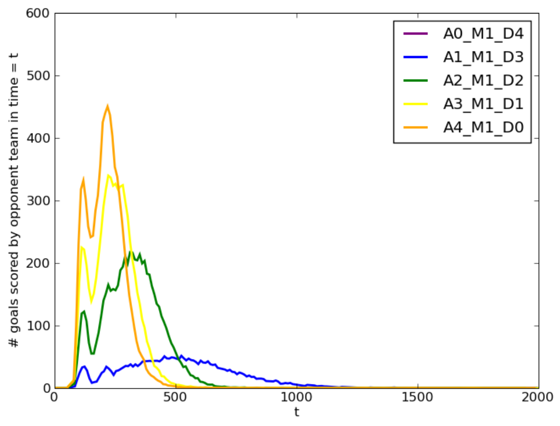


(b) Time-to-score distribution for the red team.

Figure 5.4: Cumulative time-to-score distributions for (a) the blue team and (b) the red team when the blue team plays `A3_M1_D1`, for each possible red play.



(a) Time-to-score distribution for the blue team.



(b) Time-to-score distribution for the red team.

Figure 5.5: Instantaneous time-to-score distributions for (a) the blue team and (b) the red team when the blue team plays `A3_M1_D1`, for each possible red play. This is the same data as Figure 5.4, but plotted as an (unnormalized) probability mass function rather than as a cumulative distribution function.

Figure 5.1 shows the time-to-score distributions for (a) the blue team and (b) the red team when the blue team plays A2_M1_D2, for each possible red play. The x-axis shows the amount of time elapsed t and the y-axis shows the number of points scored by the appropriate team in time $\leq t$. (These graphs essentially show the cumulative distribution function of each team's time-to-score, except that the probabilities have not been normalized to 1. By not normalizing to 1, we highlight the difference in *number* of goals scored for each team in each play combination.) Figure 5.1(a) shows that the blue team scores 320,087 points when red plays A5_M0_D0, and that these points are scored very quickly—the median time elapsed for a blue score is 109 time steps, and the 90th percentile is 156 time steps. The blue team's high scoring makes intuitive sense: since the red play assigns no defenders at all, it should be quite easy for the blue team to score. If red plays A4_M1_D0 instead, the single midfielder makes it significantly harder for blue to score. Blue scores 82,249 points when red plays A4_M1_D0, and the median time-to-score is 249 time steps (90th percentile: 372 time steps).

When the red team plays A5_M0_D0, the number of points scored by the blue team is approximately four times greater than for any other case. It is therefore difficult to distinguish the difference between the other plays in Figure 5.1. Figure 5.2 shows the same exact data as Figure 5.1, but with the y-axis zoomed in, so that the differences between plays are shown more clearly. Figure 5.2(a) shows that, as the red team adds more defenders, the number of points scored by the blue team decreases and the amount of time needed to score increases. However, there is not a significant difference in blue's score between red's plays A1_M1_D3 and A0_M1_D4.

Figure 5.2(b) shows the time-to-score distribution for the red team, when the blue team plays A2_M1_D2, for each possible red play. When red plays A4_M1_D0, the red team scores 41,481 points, and the median time-to-score is 276 time steps (90th percentile: 468 time steps). Figure 5.2(b) generally shows that as the red team uses more attackers, the number of scores made by red increases. However, there is one exception: the red team scores fewer points when playing A5_M0_D0 (11,666) than when playing A4_M1_D0 (41,481). This reduction in score can be explained by the fact that in this case, red has no defenders at all; therefore blue's two attackers can easily capture red's flag. The only way red can score is if red's five attackers capture blue's flag (defended by a blue midfielder) faster than blue captures red's undefended flag. Even though red scores fewer points overall with A5_M0_D0, most of the points that are scored happen more quickly than for any other choice of red play (median: 110 time steps, 90th percentile: 210 time steps). Therefore, it still might make sense for red to play A5_M0_D0 if red is behind and there is very little time left in the game. In the next section, we present the optimal policies for CTF; these policies will show that that there are indeed cases in which the optimal action is A5_M0_D0. Figure 5.2 further shows that the time-to-score curves for the blue team and the red team are almost identical when the red team plays A2_M1_D2; since both blue and red are playing the same play, these two distributions are nearly identical.

Figure 5.3 shows the same data as Figure 5.2 (the time-to-score distributions for each team when the blue team plays `A2_M1_D2`), but plotted as an (unnormalized) probability mass function, rather than as a cumulative distribution function. That is, for each value of t on the x-axis, the y-axis gives the number of goals scored by the given team in time exactly equal to t. For example, we can see from this graph that, when the red team plays `A4_M1_D0`, the blue team scores most of its goals in the range of 250–300 seconds (though there is another peak earlier, at approximately $t = 100$.) As the red team chooses more defensive plays, the blue team scores fewer goals, and each goal generally takes longer. When the red team plays `A5_M0_D0`, the blue team scores over 7000 goals at approximately $t = 100$. The lines for red's choice of `A5_M0_D0` have therefore been removed from the graphs; otherwise the scale of the y-axis would make it impossible to distinguish the other lines.

Figures 5.4 and 5.5 show the same data as Figures 5.2 and 5.3, but with the blue team playing `A3_M1_D1` instead of `A2_M1_D2`. The differences between `A3_M1_D1` and `A2_M1_D2` are mostly straightforward: against any given red play, blue scores more points with `A3_M1_D1`, but also gives up more points to the opponent. One interesting result is that blue scores significantly more points against `A0_M1_D4` than against `A1_M1_D3`, even though `A0_M1_D4` assigns more red defenders. There are two reasons that contribute to this result: 1) the amount of additional protection afforded by adding one more defender is minimal and 2) the presence of an attacker in `A1_M1_D3` means that red has the opportunity to score points, and if both blue and red are carrying the flag but red scores first, blue is prevented from achieving a score.

We have shown here just a sample of the time-to-score distributions—two different plays for the blue team against all six possible plays for red. Section B.2 in Appendix B shows the complete time-to-score data for all 36 combinations of blue and red plays.

5.3.3 CTF Optimal Policies

In this section, we present the optimal policies against each of the six possible opponent plays. These policies are computed by the TRSMDP optimal solution algorithm presented in Section 5.2, given the time-to-score distributions measured in the previous section.

Figure 5.6 shows the optimal policy for the CTF domain, assuming that the opponent plays `A2_M1_D2` for the entire game. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference). The colored areas show the

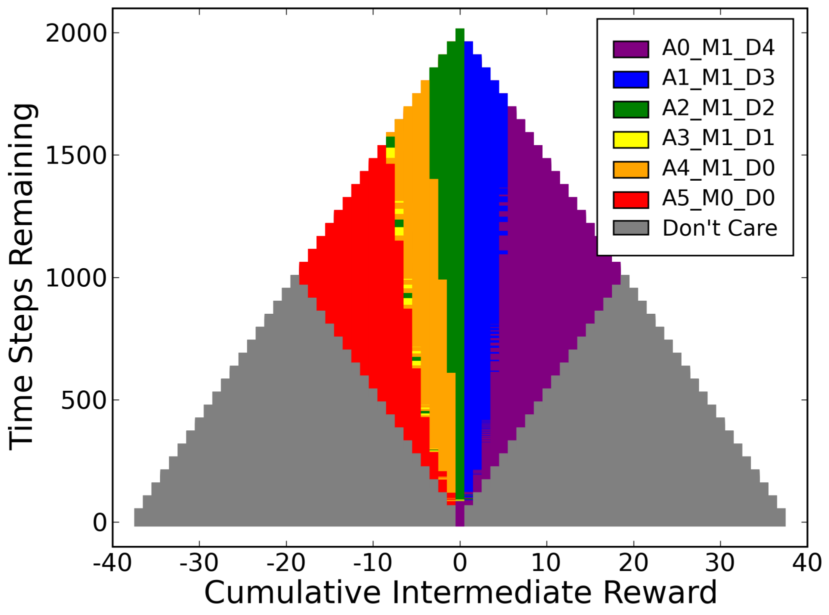


Figure 5.6: The optimal policy for the CTF domain, assuming that the opponent plays `A2_M1_D2` for the entire game. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference).

optimal action for every possible combination of time remaining and intermediate reward.[1] This policy has an expected reward of 0.1307. If our team follows this policy, we expect that $P(\text{winning}) - P(\text{losing}) = 0.1307$.[2]

Qualitatively, the optimal policy chooses `A2_M1_D2` as long as the game is tied. If our team is winning by one or more points, the optimal policy is to choose a more defensive play: usually `A1_M1_D3`, but possibly `A0_M1_D4` if our team has a very big lead or if it is very close to the end of the game. For example, our team will play `A0_M1_D4` if we are ahead by four or more points with 500 time steps remaining. If our team is losing by a point or two early in the game, our team continues to play `A2_M1_D2`, but if our team is losing by a significant amount or if it is very close to the end of the game, we will play a more aggressive play such as `A4_M1_D0` or

[1]Note that our team can switch plays at any point, not only after a point has been scored.

[2]Note that, compared to Figure 3.3, which shows the optimal policy for the MDP M presented in Chapter 3, the axes in Figure 5.6 are significantly distorted. Figure 3.3 is 120 time steps "tall" and 120 intermediate-reward values "wide" in each direction, while Figure 5.6 is 2000 time steps "tall" and only 37 units "wide" since the maximum number of scores possible in a 2000-step CTF game is 37 (as discussed in Section 5.2).

A5_M0_D0. For example, our team will play A4_M1_D0 if behind by one point with 500 time steps remaining or if behind by two points with 900 time steps remaining. There are very few states in which A3_M1_D1 is the optimal play selection; the optimal policy switches from A2_M1_D2 immediately to A4_M1_D0 when the team is losing. As the time remaining decreases, the score difference needed to switch from A2_M1_D2 to a more defensive or aggressive play decreases.

As in Figure 3.3, there are "Don't Care" regions in the lower-left and lower-right of Figure 5.6. These are states from which the actions of the agent no longer have any effect on the outcome, because it is impossible for either team to score enough points to change the eventual outcome (remember from Section 5.2 that at least 53 time steps must occur in between each score.) In these regions, all actions have equal expected value, so the team can choose any play.

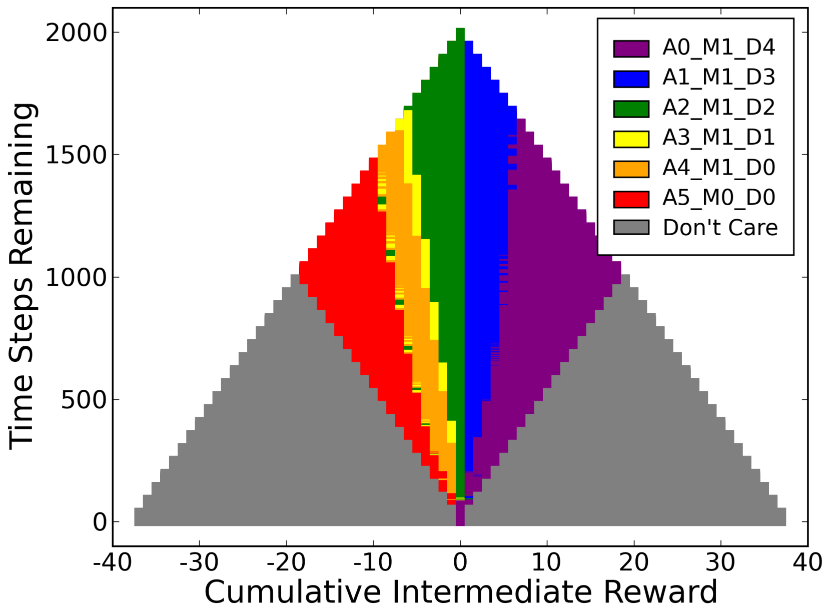


Figure 5.7: The optimal policy for the CTF domain, assuming that the opponent plays A3_M1_D1 for the entire game. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference).

Figure 5.7 shows the optimal policy for the CTF domain, assuming that the opponent plays A3_M1_D1 for the entire game. This policy has an expected reward of 0.5320. Again, the optimal policy is to play A2_M1_D2 as long as the game is tied. If our team is winning, our team may switch to a more defensive play; if our team is losing, our team may switch to a more offensive play. Compared to playing against A2_M1_D2, our team needs to be winning or losing by *more* (at a given time step) to switch from A2_M1_D2 to some other play. Also, there is a narrow range of score/time values for which A3_M1_D1 is the optimal play. This policy makes significant use of all six plays available to our team.

The optimal policies against the other four possible opponent plays are presented in Section B.3 of Appendix B. Table 5.3 shows the value of the optimal policies against each of the six possible opponents. In each case, the values are positive, meaning that we expect to win more games than we lose. The lowest value in the table is when we play against A1_M1_D3. This value is low because A1_M1_D3 is a very effective defensive play. Assuming that the score of the game is tied, the optimal play in response is A2_M1_D2; however, on average A2_M1_D2 only scores 0.517 points per game against A1_M1_D3. Therefore, games against A1_M1_D3 will be very low-scoring; a majority of these games end in 0–0 or 1–1 ties. Despite the low scores, we still expect that $P(\text{winning}) - P(\text{losing})$ is about 6% when the opponent plays A1_M1_D3. As stated above, the value of the optimal policy against A2_M1_D2 is 0.1307; against every other play, the value of the optimal policy is quite large (above 0.5).

Opponent Play	Value
A0_M1_D4	0.8179
A1_M1_D3	0.0592
A2_M1_D2	0.1307
A3_M1_D1	0.5320
A4_M1_D0	0.6286
A5_M0_D0	0.9998

Table 5.3: Values of the optimal policies against each of the six possible opponents. These values are computed theoretically, using Algorithm 5.1 to find the value of the initial state.

5.3.4 Experimental Results

We empirically validate the performance of these optimal policies against each of the six possible opponents. Our team plays 3000 CTF games against each of the six opponents, playing the optimal policy in response to each opponent. Table 5.4 presents the results. Each row of the table shows the number of wins, losses, and ties achieved by our team and the measured value of playing the optimal policy: (# wins − # losses) / 3000. Table 5.5 shows score data from the same games. Each row of the table shows the mean score achieved by our team per game, the mean score achieved by the opponent team per game, the mean score difference per game (our score minus the opponent team's score), and the theoretical score difference predicted when our team follows the optimal policy.

Opponent Play	Wins	Losses	Ties	Value
A0_M1_D4	2465	0	535	0.8217
A1_M1_D3	935	775	1290	0.0533
A2_M1_D2	1185	946	869	0.0797
A3_M1_D1	1815	511	674	0.4347
A4_M1_D0	2035	467	498	0.5227
A5_M0_D0	2998	0	2	0.9993

Table 5.4: Results of playing 3000 CTF games against each of the six possible opponents. Each row shows the number of wins, losses, and ties achieved by our team, and the measured value of playing the optimal policy.

Opponent Play	Our Score	Opponent Score	Score Diff.	Predicted Score Diff.
A0_M1_D4	1.8627	0.0000	1.8627	1.6435
A1_M1_D3	0.5123	0.7630	-0.2507	-0.2349
A2_M1_D2	1.2447	1.5323	-0.2876	-0.1639
A3_M1_D1	2.3260	1.5013	0.8247	0.9413
A4_M1_D0	3.2893	1.9013	1.3880	1.5282
A5_M0_D0	17.5053	1.3910	16.1143	14.0182

Table 5.5: Results of playing 3000 CTF games against each of the six possible opponents. Each row shows the mean score per game for our team and the opponent team, the mean score difference per game, and the theoretical score difference predicted by the model.

Against A1_M1_D3 and A2_M1_D2, our team scores fewer points per game than the opponents, but still achieves more wins than losses. This is because the optimal policy sacrifices expected intermediate reward in order to maximize the probability of winning, playing conservatively when winning and aggressively when losing. Playing conservatively when winning means that our team does not tend to win by a large margin; playing aggressively when losing means that our team tends to lose by a large margin. For example, against A1_M1_D3, our team outscores the opponent by 1.04 points (on average) for each winning game; the opponents outscore our team by 2.22 points (on average) for each losing game.

The empirical values of the policies against A0_M1_D4, A1_M1_D3, and A5_M0_D0 are very close to the theoretical values presented in Table 5.3. Against the other three plays, the empirical values are lower than the predicted values by 0.051 (for A2_M1_D2) to 0.106 (for A4_M1_D0). This minor discrepancy between the theoretical and experimental values is explained by the fact that the TRSMDP model assumes that the new time-to-score distributions take effect immediately when the team decides to switch from one play to another. In reality, it takes some time for the team members to reconfigure themselves into their new roles (e.g., moving to their new positions), so the gain realized by switching plays is slightly less than the predicted value.[3] The cost of switching plays can also be seen from the score data in Table 5.5; when the opponent plays with 1–4 attackers, the predicted score difference is slightly higher than the actual score difference. Despite this unmodeled cost of switching plays, the value of playing the optimal policy is still positive against each opponent, indicating that there is still a significant benefit to switching plays based on the score of the game and the time remaining.

5.4 TRSMDPs Applied to the Robot Soccer Domain

In this section, we apply the TRSMDP solution algorithm to find an optimal policy for a game of real robot soccer—though in a slightly simplified setting. Due to the labor-intensive nature of gathering time-to-score distributions for robot soccer, we only consider two different plays here. See Section 2.2 for details of the full robot soccer domain; our modifications to this domain are presented in Section 5.4.1. Section 5.4.2 presents the time-to-score distributions for each of the two plays. Section 5.4.3 presents the optimal policy for the robot soccer domain.

5.4.1 Experimental Domain

For the results presented in this section, we use a domain based on the official RoboCup Four-Legged League rules for 2008 [56]. In each experiment, two teams of three robots play a 10-minute-long game of soccer. Both teams are from Carnegie Mellon's CMDash'08 entry to the RoboCup 2008 US Open, which is an improvement of the CMDash'07 team that took third place (out of 24 teams) in the 2007 world competition [13]. Therefore, the underlying behaviors and skills of each team are identical; the only difference between experimental trials is the selection of plays for each team. For our experimental trials, we made the

[3]This cost of switching plays also has a pronounced effect in robot soccer, since a robot requires 20–30 seconds to walk from one end of the field to the other. As a result, we designed our play system for robot soccer to be resistant to oscillation in play choice, and to allocate roles in a manner that minimizes the amount of time needed for team reconfiguration. Further details of these algorithms used by our robot soccer team are presented in Appendix A. In the CTF domain, an attacker moving to defense may need about 50 time steps to move to its new position; 50 steps is 2.5% of the total game length, which is comparable to 30 seconds of robot soccer.

following changes to the official rules:

- Each team consisted of three field player robots (rather than four) and no goalkeepers.
- Due to the lack of goalkeepers, the illegal defender rule (which prevents the defending team's field players from entering the goal box) was not enforced.
- All other penalties, such as player pushing, leaving the field, and ball holding, were enforced. In case of a penalty, the offending player(s) were picked up and moved immediately to the halfway line (a "0-second standard removal penalty") rather than remaining out of play for 30 seconds as specified in the rules.

We made these changes so a single human referee could successfully judge a game. (In competition matches, four humans are required to referee a game.)

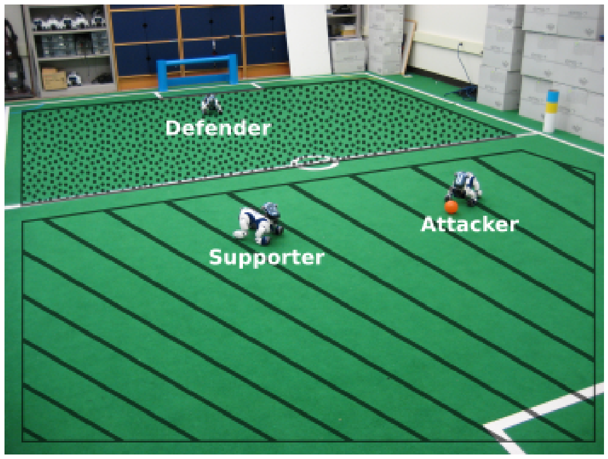
We focus here on two separate plays: RoboCup and SuperDefense. The RoboCup play is the default play we have generally used in previous competitions [74]. This play assigns three roles:

- an *attacker* robot that chases the ball over the entire field;
- a *defender* robot that protects the defensive area of the field; and
- a *supporter* robot that stays in the offensive region of the field.

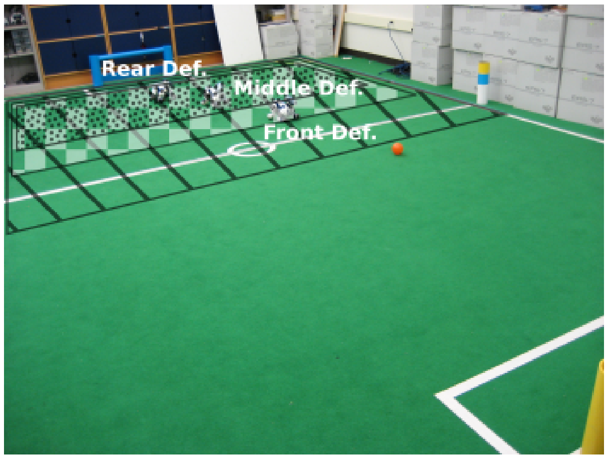
The RoboCup play is a balanced strategy that allows our team to score effectively and also provides a reasonable defense. In contrast, the SuperDefense play assigns all three field players to defensive roles:

- a *front defender* robot that covers approximately the same region as a normal defender robot, but stands much further forward when the ball is not in the defensive half of the field;
- a *middle defender* robot that covers a slightly smaller region and usually stands about a meter behind the front defender; and
- a *rear defender* robot that covers the area closest to the goal and usually stands about a meter behind the middle defender.

Figure 5.8 shows the regions covered by each of the roles in the two plays. Typically, when playing SUPERDEFENSE, the front defender will immediately engage the ball when the ball enters the defensive half of the field. If the front defender fails to clear the ball, the ball will eventually enter the middle defender's region; the middle defender will then join the front defender in trying to clear the ball. If the attacking team is still able to advance the ball forward near the goal, the rear defender joins in the defense and also attempts to clear the ball. By employing this strategy of defense-in-depth, the SUPERDEFENSE play is intended to make it very difficult for an opponent to successfully score a goal. However, the SUPERDEFENSE play comes with a significant drawback; namely, it is very unlikely that this play will ever score a goal. In a real game situation, our team would never run the SUPERDEFENSE play for an entire game; rather, this is a strategy we would like to employ when our team is ahead near the end of the game and we wish to prevent the opponents from scoring an equalizer goal.



(a) ROBOCUP play.



(b) SUPERDEFENSE play.

Figure 5.8: Regions covered by each role in each play. (a) ROBOCUP play. The defender's region is colored with dark dots; the supporter's region is colored with diagonal lines. The attacker's region is the entire field. (b) SUPERDEFENSE play. The rear defender's region is colored with dark dots; the middle defender's region is colored with a light checkerboard pattern, and the front defender's region is colored with diagonal lines.

5.4.2 Robot Soccer Time-To-Score Distributions

In this section, we aim to answer the question: do the ROBOCUP and SUPERDEFENSE plays actually produce significantly different outcomes when run against an opponent team? In addition, we measure the time-to-score distributions of RoboCup and SuperDefense. To measure these distributions, our team plays a series of games against a static team configured to use the RoboCup play. For the control case, our team also plays the RoboCup play. For the experimental case, our team plays the SuperDefense strategy. We are interested in seeing if the SuperDefense play leads to fewer opponent goals than RoboCup, and also if the time between opponent goals is significantly higher for SuperDefense. Each case is tested in 12 independent 10-minute games, for a total of 4 hours' worth of game time. To ensure fairness between the trials, the colors of the robots' uniforms and the sides of the field are swapped between each trial. The final score of each game and the time of each goal was recorded.

When playing RoboCup against RoboCup, our team scored 35 goals in total; this is a mean of 2.93 goals per game. The opponent team scored 42 goals in total, a mean of 3.50 goals per game. When playing SuperDefense against RoboCup, our team scored no goals; the opponent team scored a total of 22 goals, a mean of 1.83 goals per game. Figure 5.9 shows a histogram of the number of goals scored by our opponents in each of the games.

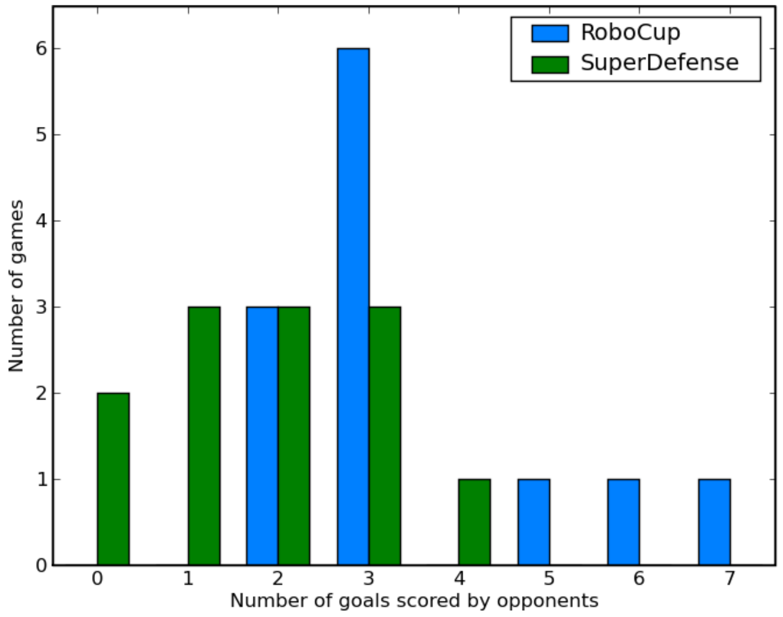


Figure 5.9: Histogram of the number of goals scored per game for the two conditions. When our team uses the SuperDefense play, the opponents score significantly fewer goals.

We test the statistical significance of the difference between the number of opponent goals scored per game using Student's two-tailed *t*-test (assuming unequal variances) and a one-way analysis of variance (ANOVA). Both tests indicate that the number of opponent goals scored when we play SuperDefense is significantly fewer than the number of opponent

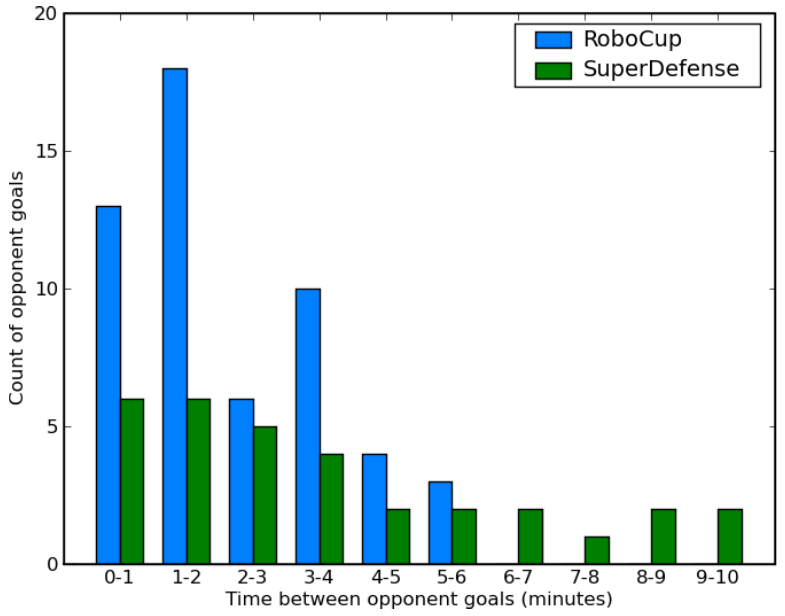


Figure 5.10: Histogram of time between opponent goals. When our team uses the SuperDefense play, the opponents take significantly longer to score goals.

goals scored when we play RoboCup (for the t-test, $p = 0.0107$; for the ANOVA, $F(1, 23) = 7.8571$, $p = 0.0104$).

In addition to the number of goals scored by our opponents, we are also interested in the time-to-score distributions: how long it takes for each team to score. We therefore calculate the time between consecutive opponent goals in each of the games. At the end of the game, we pessimistically assume that the opponents score immediately after the game is over, since we do not know how long it would have taken for the opponents to score their next goal (unless the game time remaining at the time of the last opponent goal was less than 29 seconds, which was the fastest time between goals in any trial). This leads to a mean of 133.3 seconds between goals for the RoboCup condition, and a mean of 223.4 seconds between opponent goals for the SuperDefense condition. Figure 5.10 shows a histogram of the time between opponent goals for the two conditions.

We again test the statistical significance of the difference between the distributions using Student's two-tailed t-test (assuming unequal variances) and a one-way ANOVA. Both tests indicate that the amount of time between opponent goals when we play SuperDefense is significantly greater than the amount of time between opponent goals when we play RoboCup (for the t-test, $p = 0.0077$; for the ANOVA, $F(1, 85) = 10.7345$, $p = 0.0015$).

Taken together, these results show that the SuperDefense play successfully hinders the opponent team from scoring. Compared to RoboCup, the SuperDefense play significantly reduces the number of goals scored by the opponent and also increases the amount of time between opponent goals. However, the SuperDefense play completely prevents our own team from scoring. In the next section, we present the policy which changes strategy in order to maximize the probability of winning in the robot soccer domain.

5.4.3 Robot Soccer Optimal Policy

In this section, we present the optimal policy for the robot soccer domain. This policy is computed by the TRSMDP optimal solution algorithm presented in Section 5.2, given the time-to-score distributions measured in the previous section.

Figure 5.11 shows the optimal policy for the robot soccer domain, assuming that the opponent plays RoboCup for the entire game. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference). The y-axis ranges from 0 to 600 because our games are ten minutes long in this domain and our time resolution is one second. The colored areas show the optimal action for every possible combination of time remaining and intermediate reward. This policy has an expected reward of 0.0583. If our team follows this policy, we expect that $P(\text{winning}) - P(\text{losing}) = 0.0583$.

The optimal policy chooses the RoboCup play if the game is tied or if our team is losing. If our team is winning, the optimal policy is to play SuperDefense instead. In this case, the choice to switch to SuperDefense does not depend on how much our team is winning by, or on how much time is remaining in the game.

Compared to the CTF results, our robot soccer results are based on relatively little time-to-score data: 22 to 42 goals per distribution, compared to over 4500 points in the lowest-scoring CTF configuration. The time-to-score distributions we measured are a rough approximation to the true distributions; we therefore expect that the optimal policy, and its value, might change somewhat if we measured more data. However, the difficulties of collecting more data for the RoboCup domain are twofold: first, these games all need to be refereed by a human, so it is very time-consuming to run additional trials; second, the AIBO robots, while fairly robust, are not designed to handle playing for such extended periods of time—three robots were broken during the course of these experiments. For these same reasons, we were unable to run the optimal policy on the real robots for extensive verification, as we did for the CTF domain. We therefore treat the robot soccer results as a proof-of-concept demonstration of our algorithms on our real robot soccer team. Our more exhaustive CTF results further demonstrate that there is significant value to reasoning about score and time in complex

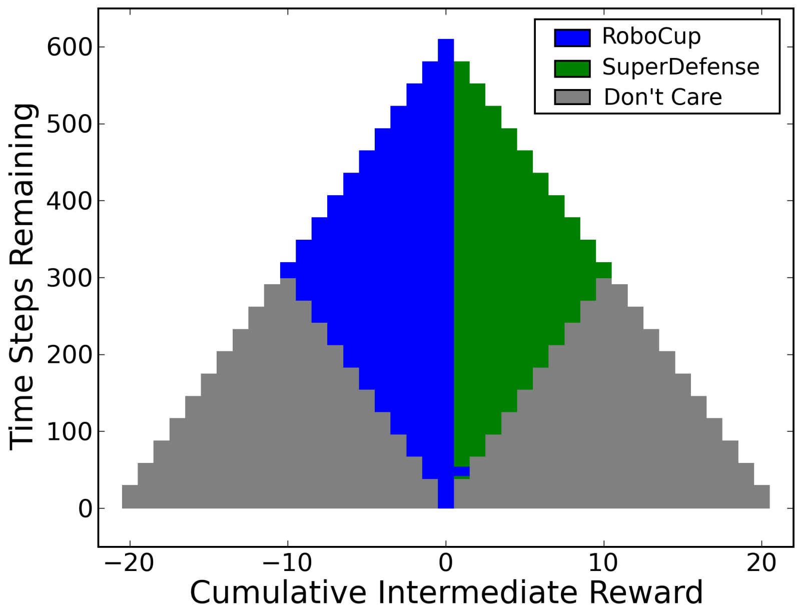


Figure 5.11: The optimal policy for the robot soccer domain, assuming that the opponent plays RoboCup for the entire game. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference).

adversarial environments.

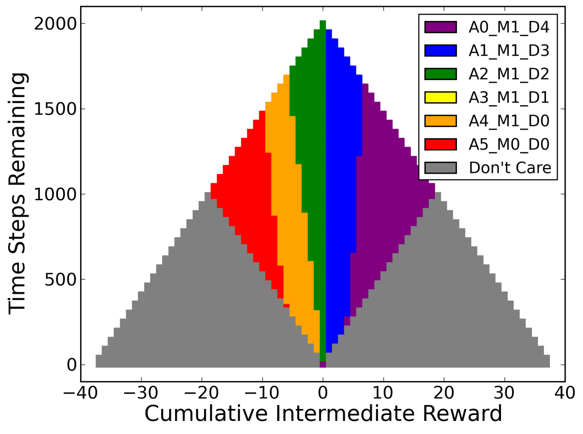
5.5 Comparison Between MDP and SMDP Approaches

Our motivation for choosing SMDPs to model the Capture the Flag domain is that the time between state transitions (and/or reward assignments) in an MDP model is drawn from a geometric distribution. From the time-to-score distributions presented in Section 5.3.2, it is clear that the actual time-to-score distributions for Capture the Flag do not in fact follow a geometric distribution. However, we have not yet shown whether modeling the time-to-score distributions accurately actually leads to increased performance in the CTF domain. In this section, we use an MDP-based approach to derive optimal policies for the CTF domain, and empirically compare the effectiveness of the resulting policies with the SMDP approach.

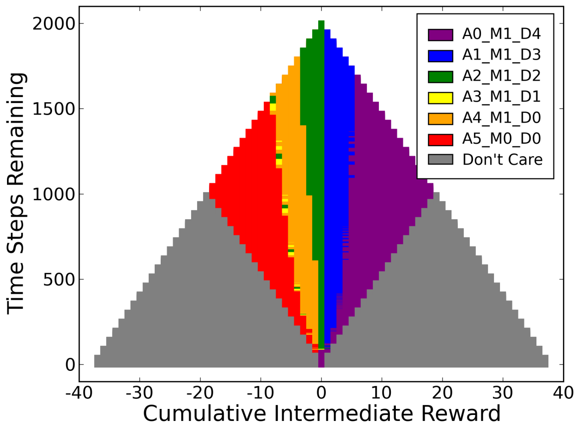
Our domain model for the MDP approach is relatively simple. We again choose to represent the domain as a single state. We still need to measure the probabilities of scoring a goal for each of the 36 possible play combinations (a_b, a_r). To generate these probabilities, we use the same data collected in Section 5.3.2, in which we ran each play combination for 40 million time steps in the CTF simulator. For each play combination, we count the total number of blue goals scored and red goals scored, and divide each number by 40 million to find the per-step probability of scoring a goal for each team. Formally, if the blue team scored x goals and the red team scored y goals when (a_b, a_r) was played, the reward distribution is as follows:

$$R(s_0, (a_b, a_r)) = \begin{cases} +1 & (p = \frac{x}{40000000}) \\ -1 & (p = \frac{y}{40000000}) \\ 0 & (p = 1 - \frac{x+y}{40000000}) \end{cases} \tag{5.1}$$

Given the reward distributions, we run the TRMDP solution algorithm (presented in Chapter 3) to find optimal policies against each possible opponent play. Figure 5.12(a) shows the optimal policy for the CTF domain, assuming that the opponent plays `A2_M1_D2` for the entire game. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference). The colored areas show the optimal action for every possible combination of time remaining and intermediate reward. This policy has an expected reward of 0.1277. If our team follows this policy, we expect that $P(\text{winning}) - P(\text{losing}) = 0.1277$. For comparison, Figure 5.12(b) shows the optimal TRSMDP policy against `A2_M1_D2` (this figure is identical to Figure 5.6).



(a) Optimal MDP policy.



(b) Optimal SMDP policy.

Figure 5.12: Optimal policies for the CTF domain, assuming that the opponent plays `A2_M1_D2` for the entire game. (a) Optimal policy generated by using an MDP as the base model. (b) Optimal policy generated by using an SMDP as the base model.

The MDP policy has the same general form as the SMDP policy, but a close look reveals significant differences. The MDP policy is more conservative than the SMDP policy, requiring a greater score difference to change from `A2_M1_D2` to a more conservative or aggressive play. For example, when there are 500 time steps remaining, the SMDP policy will play `A4_M1_D0` if our team is losing by a single point, but the MDP policy will play `A2_M1_D2` only if our team is losing by two or more points. A similar trend is seen when winning as well: when there are 500 time steps left, the SMDP policy will play `A0_M1_D4` if our team is winning by four or more points, but the MDP policy will play `A0_M1_D4` only if our team is winning by five or more points.

Opponent Play	TRMDP Value	TRSMDP Value
`A0_M1_D4`	0.8412	0.8179
`A1_M1_D3`	0.0653	0.0592
`A2_M1_D2`	0.1278	0.1307
`A3_M1_D1`	0.5231	0.5230
`A4_M1_D0`	0.6227	0.6286
`A5_M0_D0`	1.0000	0.9998

Table 5.6: Theoretically-computed values of the optimal policies against each of the six possible opponents.

Opponent Play	Wins	Losses	Ties	TRMDP Value	TRSMDP Value
`A0_M1_D4`	2389	0	611	0.7963	0.8217
`A1_M1_D3`	841	774	1385	0.0223	0.0533
`A2_M1_D2`	1095	927	978	0.0560	0.0797
`A3_M1_D1`	1835	614	551	0.4070	0.4347
`A4_M1_D0`	2053	515	432	0.5127	0.5227
`A5_M0_D0`	2999	0	1	0.9997	0.9993

Table 5.7: Results of playing 3000 CTF games against each of the six possible opponents. Each row shows the number of wins, losses, and ties achieved by our team, and the measured value of playing the optimal TRMDP and TRSMDP policies.

Table 5.6 shows the theoretically-computed values of the optimal policies against each of the six possible opponents for both the MDP and SMDP models. The values are generally close, and neither approach consistently provides a higher value. However, these models are only approximate—especially the MDP model, which implicitly assumes that the time-to-score distributions are geometric. What we are really interested in is the *empirical* performance of the two different approaches—how well each approach does in practice. To test the performance of the TRMDP approach, we ran 3000 CTF games against each of the six opponents, playing the optimal TRMDP policy in response to each opponent. Table 5.7 presents the results. Each row of the table shows the number of wins, losses, and ties achieved by our team and the measured value of playing the optimal TRMDP policy: (# wins − # losses) / 3000. For comparison, the empirical values of the TRSMDP policies are also shown; these values are identical to those shown in Table 5.5. Table 5.7 shows that the TRSMDP approach outperforms the TRMDP approach for every opponent play. Though the absolute difference between the values is relatively small, the percentage difference is relatively large for the opponents which are difficult to beat (`A1_M1_D3` and `A2_M1_D2`). The SMDP approach allows for increased accuracy in modeling the true time-to-score distributions of the Capture the Flag domain, which leads to better empirical performance against each opponent.

5.6 Threshold-Plus-Linear Objective Function

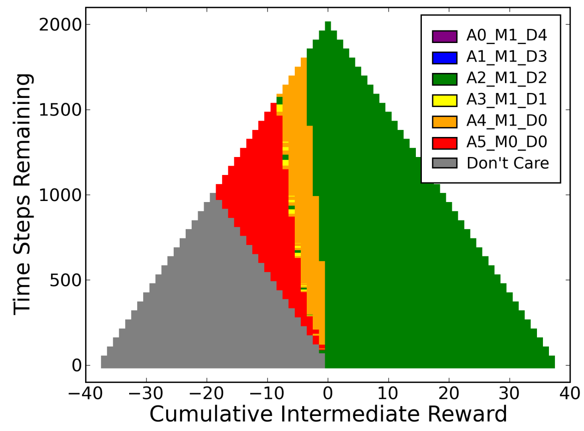
All the results previously presented in this thesis utilize the zero-sum threshold function (Equation 3.1). We have argued that the zero-sum threshold function is the best objective function for a team that desires to maximize the probability of winning (rather than maximizing the score difference). However, in some real-world situations (such as a qualifying round for a tournament), winning is of *primary* importance, but winning by a large margin is also desired. In such domains, we desire an objective function that is a mix of a threshold function and a function that is linear in the score difference. In this section, we consider one example of such an objective function, applied to the Capture the Flag domain.

We define the *threshold-plus-linear-k* (TPL-k) objective function as follows:

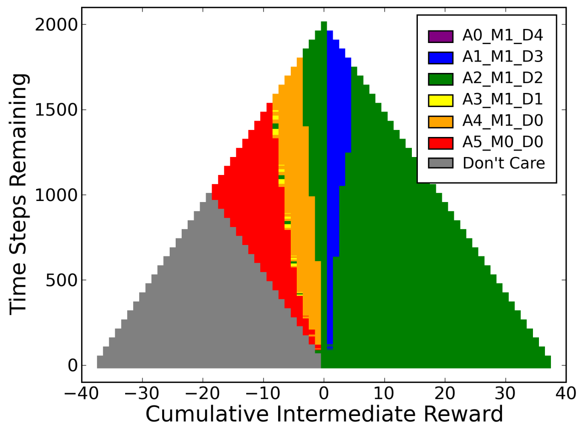
$$r_{true} = \begin{cases} k + r_{intermediate} - 1 & \text{if } r_{intermediate} > 0 \\ 0 & \text{if } r_{intermediate} = 0 \\ -k & \text{if } r_{intermediate} < 0. \end{cases} \tag{5.2}$$

In this scenario, a loss counts as $-k$ reward (regardless of the magnitude of the loss) and a tie counts as 0 reward. A win by a single point counts as k reward, with an additional point of reward assigned for each additional point in the margin of victory. k trades off between the value of winning and the value of the margin of victory. If k is large, winning is weighted heavily; if k is small, the margin of victory is weighted heavily.

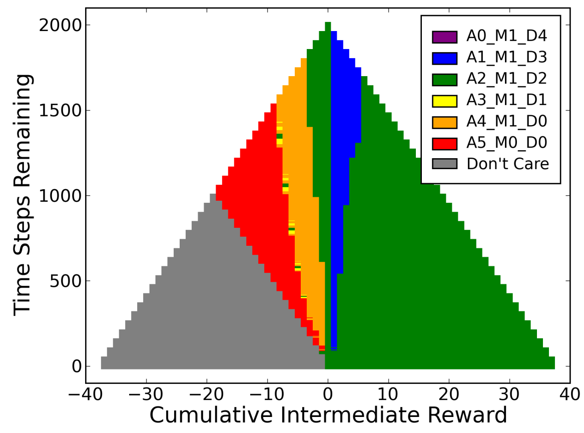
Given the CTF time-to-score distributions and the TRSMDP solution algorithm, we can find optimal policies for the threshold-plus-linear-k objective function. Figure 5.13(a) shows the



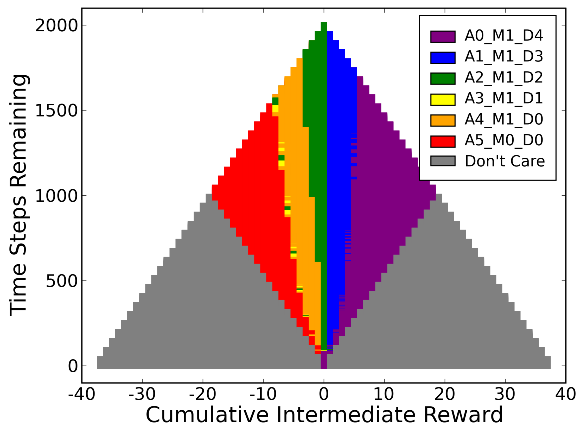
(a) Threshold-plus-linear-1 threshold function.



(b) Threshold-plus-linear-5 threshold function.



(c) Threshold-plus-linear-10 threshold function.



(d) Zero-sum threshold function.

Figure 5.13: Optimal CTF policies against `A2_M1_D2` with the threshold-plus-linear-k threshold functions.

optimal policy for the CTF domain with the TPL-1 objective function, assuming that the opponent plays A2_M1_D2 for the entire game. Figures 5.13(b) and 5.13(c) show the optimal policies with the TPL-5 and TPL-10 objective functions, respectively. Figure 5.13(d) shows the optimal policy against A2_M1_D2 with the zero-sum threshold function (this figure is identical to Figure 5.6, and is presented again here for comparison to the other policies).

One consistent difference between the optimal zero-sum policy and the optimal TPL-k policies is that none of the TPL-k policies have a "Don't Care" region in the lower right. With the zero-sum threshold function, this region corresponds to states in which winning is guaranteed (according to the model) and therefore every action is equally valuable. However, when a win is guaranteed, the threshold-plus-linear objective function still values maximizing the score difference, so the optimal action is the play which maximizes expected score difference (A2_M1_D2).

The TPL-1 policy puts an extreme value on the margin of winning: winning by n points is valued n times as much as winning by a single point. The optimal policy shown in Figure 5.13(a) reflects this desire to "run up the score": as long as our team is tied or ahead, the optimal play choice is A2_M1_D2, which maximizes the expected score difference. There is never a case in which our team plays a more defensive play such as A1_M1_D3 or A0_M1_D4.

In contrast, the TPL-5 and TPL-10 policies put significantly more value on winning than on the margin of victory. Therefore, if our team if winning by only a single point, the optimal action is the more defensive A1_M1_D3 play, which aims to preserve the lead. However, our team will switch back to A2_M1_D2 to run up the score if our team is winning by a margin significant enough to nearly guarantee victory (such as being ahead by 2 points with less than 400 time steps remaining). The general forms of these two policies are nearly identical, but a close inspection shows that the TPL-10 policy is a bit more conservative, requiring a larger score difference before trying to run up the score. For example, when there are 1000 time steps remaining, the TPL-5 policy will play A2_M1_D2 to run up the score if we are ahead by 3 or more points, but the TPL-10 policy requires a score difference of 4 or more points to play A2_M1_D2. This increased conservatism is exactly what we would expect, since a higher value of k puts more weight on the value of winning and relatively less weight on the margin of the victory.

The TPL-k policies sacrifice the probability of winning somewhat, in order to increase the chances of winning by a large margin. This raises two obvious questions: (1) How much do we lose, in terms of our chances of winning when we play a TPL-k policy? (2) How much do we gain, in terms of the expected score difference for the games in which our team wins?

Table 5.8 answers question (1) by showing the theoretically-computed values of each policy against each opponent, according to the zero-sum reward threshold function. Every entry in the table is equal to $P(\text{winning}) - P(\text{losing})$ when the given policy is played against the given opponent. The zero-sum reward threshold function maximizes $P(\text{winning}) - P(\text{losing})$, so its value is greater than or equal to the value of every other policy. In contrast, the TPL-1 policy is very eager to drive up the score, which has the potential to cause a significant loss of value compared to the optimal policy. For example, against `A2_M1_D2`, the optimal zero-sum policy achieves a value of 0.1307, but the TPL-1 policy only achieves a value of 0.0219. In fact, TPL-1 sacrifices significant value against all opponent plays except the extreme cases of `A0_M1_D4` and `A5_M0_D0`, in which the opponent is attacking with no players or with all five players. In comparison, TPL-5 and TPL-10 do not sacrifice winning nearly as much; for example, the value of TPL-10 is within 0.02 of optimal against every opponent play.

Opponent Play	Zero-sum Value	TPL-1 Value	TPL-5 Value	TPL-10 Value
`A0_M1_D4`	0.8179	0.8179	0.8179	0.8179
`A1_M1_D3`	0.0592	0.0331	0.0582	0.0591
`A2_M1_D2`	0.1307	0.0219	0.1306	0.1307
`A3_M1_D1`	0.5320	0.4784	0.5059	0.5152
`A4_M1_D0`	0.6286	0.5947	0.6119	0.6156
`A5_M0_D0`	0.9998	0.9998	0.9998	0.9998

Table 5.8: Shows the theoretically-computed values of each policy against each opponent, according to the zero-sum reward threshold function. Every entry in the table is equal to $P(\text{winning}) - P(\text{losing})$ when the given policy is played against the given opponent.

Opponent Play	Zero-sum Score	TPL-1 Score	TPL-5 Score	TPL-10 Score
`A0_M1_D4`	2.0095	2.2866	2.2866	2.2866
`A1_M1_D3`	1.0041	1.1660	1.0298	1.0108
`A2_M1_D2`	1.3435	1.7564	1.3477	1.3468
`A3_M1_D1`	1.8784	2.7153	2.5404	2.4288
`A4_M1_D0`	2.4208	3.2916	3.1692	3.1303
`A5_M0_D0`	14.0208	16.2945	16.2945	16.2945

Table 5.9: Shows the theoretically-computed score differences for each game which results in a win for our team. Every entry in the table is equal to the mean score difference when the given policy is played against the given opponent.

Table 5.9 answers question (2) by showing the theoretically-computed score differences for each game which results in a win for our team. Every entry in the table is equal to the mean score difference when the given policy is played against the given opponent, for games in which our team wins (i.e., in which the final score difference is ≥ 1). As expected, we see the opposite trend here: the TPL-1 policy scores the highest and the zero-sum policy scores the lowest, with TPL-5 and TPL-10 striking a balance in between. TPL-1 indeed scores significantly more goals per win than the zero-sum policy against every opponent play, though at the cost of achieving fewer wins than the zero-sum policy. TPL-5 and TPL-10 have a very interesting property: these approaches significantly increase the margin of victory against opponents which are "easy" to beat (`A0_M1_D4`, `A3_M1_D1`, `A4_M1_D0`, and `A5_M0_D0`), but perform very similarly to the zero-sum policy against opponents which are difficult to beat (`A1_M1_D3` and `A2_M1_D2`). TPL-5 and TPL-10 therefore provide a good tradeoff, providing a near-optimal probability of winning against difficult opponents, while successfully running up the score against relatively easy opponents. Ultimately, the designer of the agent must decide whether the gain in score provided by a TPL-k policy is worth the slightly decreased probability of winning. In the remainder of this thesis, we return to the problem of maximizing the probability of winning (or achieving a given score threshold) rather than also considering maximizing the margin of victory.

5.7 Summary

In this chapter, we have introduced thresholded-rewards semi-Markov decision processes (TRSMDPs) to find optimal policies for the Capture the Flag and robot soccer domains. In Section 5.2, we introduced an optimal solution algorithm for TRSMDPs.

Section 5.3.1 presented an analysis of the eleven Capture the Flag plays. After eliminating each play which was strictly worse than some other play, six plays remained. Section 5.3.2 showed the time-to-score distributions for each team for all 36 possible joint play choices. Section 5.3.3 tied all these results together to present the optimal TRSMDP policies for the CTF domain. These policies were empirically tested in Section 5.3.4. Since there are many plays and play combinations considered in this chapter, we have highlighted here only a subset of the CTF results. Appendix B includes the full results of all our CTF experiments.

Section 5.4 presented an extensive empirical analysis of the performance of two different plays, RoboCup and SuperDefense, in four hours of real robot soccer games. We showed that our team's play choice has a significant effect on the opposing team's performance. Namely, when we choose the SuperDefense play, the opponents score fewer goals per game, and take longer on average to score each goal, than when we choose the RoboCup play. Both these results are statistically significant. These results indicate that the optimal play-selection strategy for our team—the policy that maximizes the probability of winning—is to play the RoboCup play by default and to switch to the SuperDefense play when our team is winning.

In Section 5.5, we used an MDP-based approach to find optimal policies for the CTF domain. We found that the SMDP-based approach outperformed the MDP-based approach in an empirical comparison.

Section 5.6 introduced the *threshold-plus-linear-k* objective function, which trades off between maximizing the probability of winning and maximizing the margin of victory. We found that setting k to a high value, such as 10, provides a near-optimal probability of winning against difficult opponents, while successfully running up the score against relatively easy opponents.

Throughout this chapter, we have assumed that the opponent is playing a single play throughout the entire game, and that we have perfect knowledge of which play the opponent is playing. In the next chapter, we relax some of these assumptions.

Chapter 6

TRMDPs with Unknown Opponents

Chapter 5 introduces thresholded-rewards semi-Markov decision processes (TRSMDPs), which model thresholded-rewards domains in which the time needed for a state transition or reward assignment follows any arbitrary distribution. The TRSMDP solution algorithm was tested in the robot soccer and CTF domains. However, we assumed that the opponent plays a single play throughout the entire game, and that our team has perfect knowledge of which play the opponent is playing. In this chapter, we relax these assumptions.

In Section 6.1, we introduce the problem of *incidental behavior recognition*. By "incidental," we mean that our team has some primary task beyond simply classifying the behavior of other agents operating in the same environment. For example, in robot soccer, any observations of the environment or opponent robots happen by "accident" as the team is primarily engaged in playing soccer. We present experiments with our robot soccer team that demonstrate that, with the right set of state features, our team can accurately classify the behavior of the opponent team.

In Section 6.2, we direct our attention to the challenge of playing against an opponent whose behavior is initially unknown. We consider a *static opponent* scenario in which the opponent is playing a fixed, but initially unknown, play throughout the entire game and a *dynamic opponent* scenario in which the opponent changes plays over the course of the game. We utilize the concept of a *default play* as a safe response to an unknown opponent. We find empirically that `A2_M1_D2` is the best default play to use in the CTF domain. We also empirically measure the effect of playing against opponents that change plays frequently or infrequently.

6.1 Incidental Behavior Recognition in Robot Soccer

We introduce the problem of *incidental behavior recognition*—recognizing the behavior of the opponent team from our own team's observations. This is an interesting filtering problem because our multi-robot team's observations of the environment and of other robots are *incidental* rather than *purposeful*. Many robotics researchers have used multi-robot teams to address tasks such as providing surveillance of a given area, tracking multiple moving targets, or recognizing the behavior of humans or robots. However, in these domains, the team's primary task is usually to monitor the environment or other agents (as in [20, 43, 60, 64–66]); in these approaches, the entire behavior of the team is driven toward successfully observing the environment. In other approaches (such as [17, 73]), the team has a primary goal other than activity recognition, but also possesses a global view of the environment, so the low-level behaviors of the team do not significantly affect the quality of the observations.

In contrast, we define *incidental behavior recognition* as a problem in which the primary task is not to observe the environment and classify the behavior of the other team; rather, the primary task is to play soccer well, and any observations of the environment or opponent robots happen by "accident" as the team plays soccer. Since the AIBO camera's field of view is very narrow—under 60 degrees—each robot has a very limited focus of attention, which is usually focused on the ball since the ball is the most important object in the environment. In particular, our RoboCup team does not explicitly track the movements of the opponent robots nor attempt to maximize the portion of the field that is viewed.

6.1.1 Approach

We assume that the opponent's play choice is unknown, but remains static throughout the entire game. We desire an online classification algorithm: at each time step of execution, the algorithm's output is the most likely opponent play choice, given the history of observations collected by our team members so far.

To achieve this goal, we utilize hidden Markov models (HMMs) to model the behavior of the other team [51]. Our team collects training data by playing multiple games against each opponent play and recording each team member's observations of the environment. These observation logs are used as labeled training data: given these logs, we use the Baum-Welch algorithm to learn a model of the environment for each distinct opponent play. Our play-recognition algorithm operates in an online fashion. At each time step, the play recognizer uses the forward algorithm to compute the likelihood of the team's entire joint observation sequence given each of the learned models. The model which maximizes the likelihood of the observation sequence is chosen as the best estimate of which play the opponent team is running.

In order to achieve good classification performance, HMMs require that the observation vectors used as input (both at training time and classification time) contain informative features. We consider a limited set of feature-selection functions which map the raw joint observations to (presumably) more informative feature vectors; the resulting feature vectors are then used as the input during training and classification.

Due to the difficulty of collecting extensive training and test sets on the real robots, we utilize leave-one-out cross-validation to evaluate the effectiveness of the play-recognition algorithm and to determine which selections of features are most effective for the robot soccer domain.

6.1.2 Experimental Setup

In our incidental behavior recognition experiments, we utilize the RoboCup and SuperDefense plays (introduced in Section 5.4.1). The RoboCup play is the default play we have generally used in previous competitions; this play assigns an attacker robot, a defender robot, and a supporter robot. The SuperDefense play is a defensive play that assigns all three robots to defensive roles. In each experimental trial, our team plays the RoboCup play against an unknown opponent playing either RoboCup or SuperDefense. The rules of the game are the same as those presented in Section 5.4.1: the official RoboCup 2008 rules, with minor modifications to enable a single human referee to successfully judge the game.

To enable distributed play recognition, each robot r_i communicates the following data once per second:

- The robot's best estimate of its own position (px_i, py_i).
- The robot's best estimate of the position of the ball (bx_i, by_i).
- The position of the last opponent robot seen (ox_i, oy_i).

All positions are communicated in global coordinates, so the team has a common frame of reference. For each of these features, each robot also broadcasts a boolean feature indicating whether the observation is considered "valid." A robot's estimate of its own position is valid iff the robot's estimate of localization error is relatively low, as determined by a threshold on the robot's localization confidence. The ball position estimate is considered valid iff the robot's own position estimate is valid *and* the ball has been seen recently with sufficiently high confidence, as determined by our high-level vision and world modeling algorithms [75]. Similarly, the estimate of opponent robot position is considered valid iff the robot's own position estimate is valid *and* the opponent robot has been seen recently and with a high vision confidence [13].

For the purposes of our experiments, the data broadcast by each robot is also sent to an offboard computer for later processing, so that we can perform leave-one-out cross-validation and further experiments in feature selection. In a real game situation, the robots would only broadcast to each other, using the team's joint observations to classify the opponent behavior online.

Figures 6.1 and 6.2 show sample data logged by the robots from a typical RoboCup game and a typical SuperDefense game (respectively). Lines indicate the positions of our team's robots, circles indicate observations of the ball, and squares indicate observations of opponent robots. The figures only include valid observations; invalid observations are not shown. Our team's goal is located on the right side of each figure; the opponents' goal is located on the left side of each figure. In each figure, the column on the left shows the observations of the robots in the first minute of the game. The column on the right shows the observations over the course of the entire 10-minute game. Each row corresponds to the observations of one of the three robots on our team. The combined observations of the three robots form the team's joint observation.

Even though the opposing SuperDefense team does not penetrate significantly into the offensive half of the field, Figures 6.1 and 6.2 show that there is still a significant overlap in observations between the RoboCup and SuperDefense games. These overlapping observation histories show that correct classification of opponent plays is not a trivial task in the robot soccer domain.

Formally, the observation of a single robot r at timestep t is:

$$o_{r,t} = \langle bx_{r,t}, by_{r,t}, bv_{r,t}, px_{r,t}, py_{r,t}, pv_{r,t}, ox_{r,t}, oy_{r,t}, ov_{r,t} \rangle, \tag{6.1}$$

where $bx_{r,t}$ and $by_{r,t}$ are the position of the ball in global coordinates as seen by robot r at time t and $bv_{r,t}$ is a boolean flag saying whether robot r considers the ball observation to be valid. Similarly, $px_{r,t}$, $py_{r,t}$, and $pv_r{,}t$ are the position and validity of robot r's own position and $ox_{r,t}$, $oy_{r,t}$, and $ov_{r,t}$ are the position and validity of the opponent most recently observed by robot r. We then define the joint observation j_t of the team at time t as the combination of the individual robots' observations at time t: $j_t = \langle o_{1,t}, o_{2,t}, o_{3,t} \rangle$. Over the course of an entire game that is t timesteps long, the robots obtain a joint observation sequence $J = \langle j_1, j_2, \ldots, j_t \rangle$.

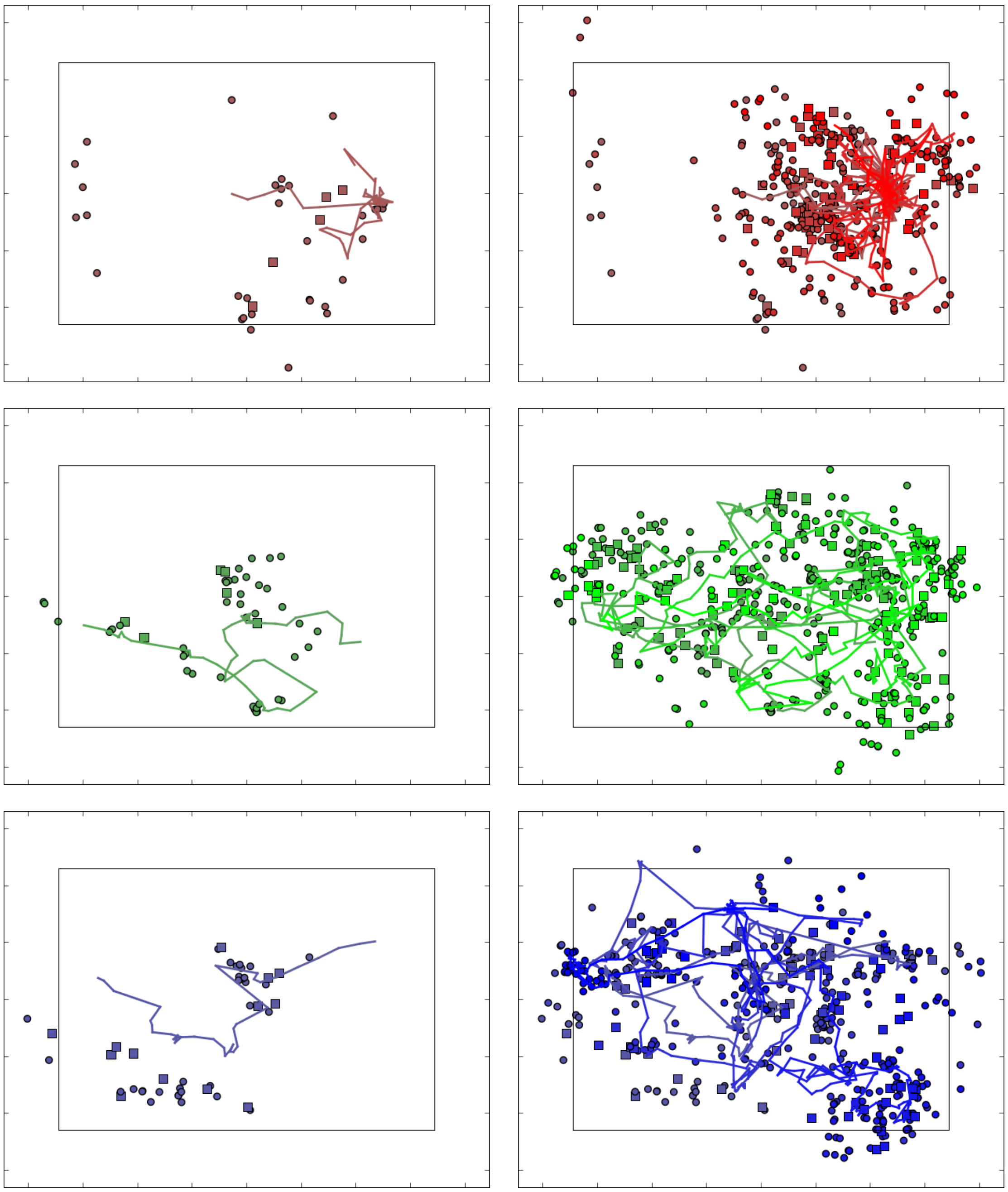


Figure 6.1: Sample log data collected by our robots when the opponents play the RoboCup play. Each image in the left column shows the observations of a single robot in the first minute of the game. Each image in the right column shows the observations of a single robot over the entire 10-minute game.

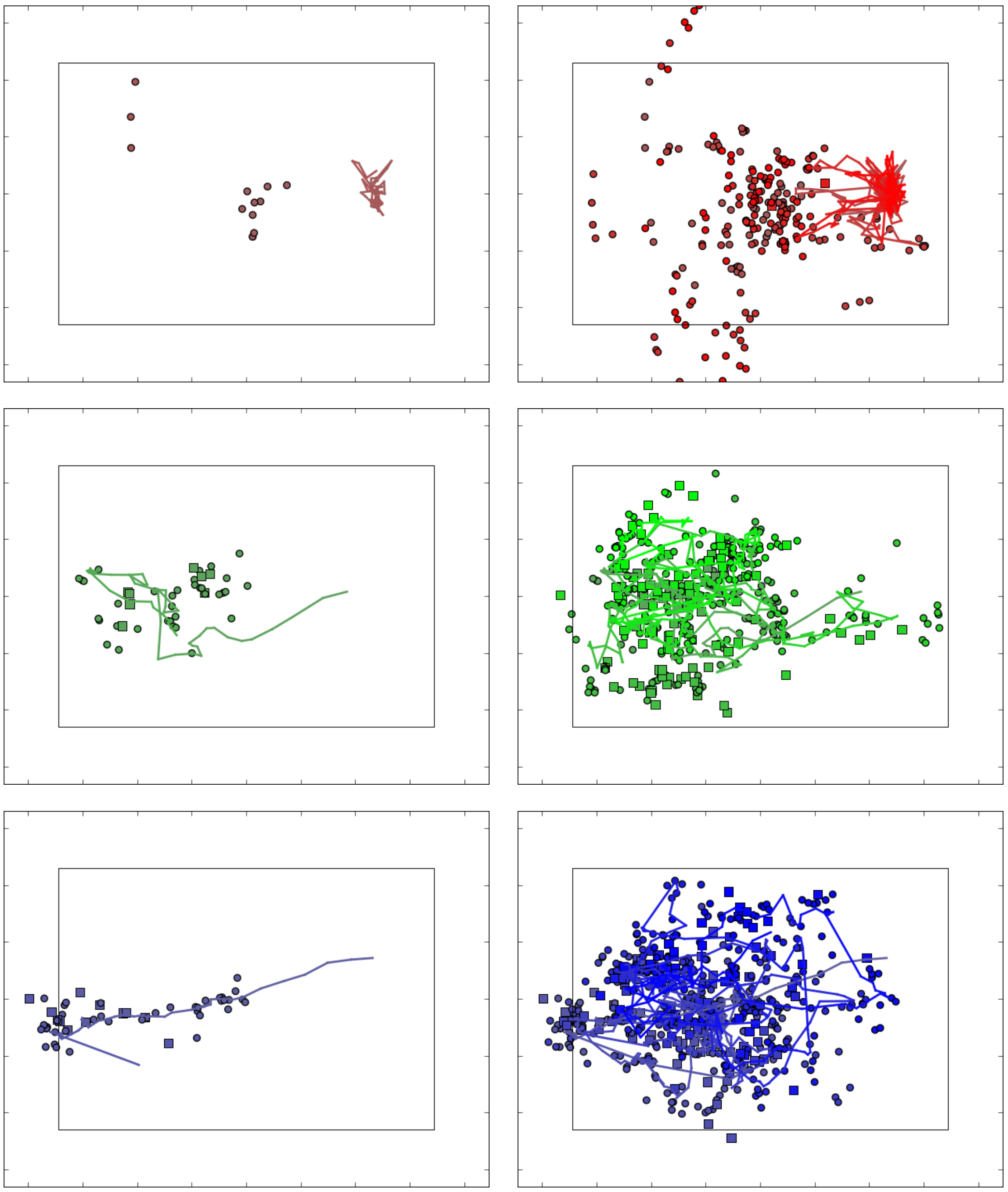


Figure 6.2: Sample log data collected by our robots when the opponents play the SuperDefense play. Each image in the left column shows the observations of a single robot in the first minute of the game. Each image in the right column shows the observations of a single robot over the entire 10-minute game.

Algorithm 6.1 PLAY-RECOGNITION algorithm.

```
 1: Given: training sets $RC_{train}$ and $SD_{train}$, observation sequence $J$ of length $t$
 2: $\lambda_{RC} \leftarrow$ BAUM-WELCH($RC_{train}$)
 3: $\lambda_{SD} \leftarrow$ BAUM-WELCH($SD_{train}$)
 4: for $i = 1$ to $t$ do
 5:     $J_i \leftarrow \langle j_1, j_2, \ldots, j_i \rangle$
 6:     $p_{RC} \leftarrow P(J_i|\lambda_{RC})$
 7:     $p_{SD} \leftarrow P(J_i|\lambda_{SD})$
 8:     if $p_{RC} \geq p_{SD}$ then
 9:         $C[i] \leftarrow$ ROBOCUP
10:     else
11:         $C[i] \leftarrow$ SUPERDEFENSE
12: return $C[i]$
```

To collect training data, we ran twelve 10-minute games against an opponent team running the ROBOCUP play and twelve 10-minute games against an opponent team running the SUPERDEFENSE play, for a total of four hours of game play time. In all trials, our own team was running the ROBOCUP play. Let RC be the set consisting of the 12 joint observation sequences gathered when playing against a ROBOCUP opponent; let SD be the set consisting of the 12 joint observation sequences gathered when playing against a SUPERDEFENSE opponent.

Algorithm 6.1 presents our play-recognition algorithm, which uses a fairly standard HMM-based approach.[1] The algorithm takes an input a labeled training set RC_{train} consisting of a set $\{J_1, J_2, \ldots, J_n\}$ of observation sequences gathered while the opponent was playing the ROBOCUP play, and another training set SD_{train} consisting of a set $\{J_1, J_2, \ldots, J_n\}$ of observation sequences gathered while the opponent was playing the SUPERDEFENSE play. On lines 2–3 of the play-recognition algorithm, the algorithm trains one HMM λ_{RC} by providing the training sequences RC_{train} as input to the Baum-Welch algorithm; the algorithm also trains another HMM λ_{SD} by providing the training sequences SD_{train} as input to the Baum-Welch algorithm. Lines 4–13 provide online classification at each timestep i. J_i is the vector of all joint observations $\langle j_1, j_2, \ldots, j_t \rangle$ seen by the team in the first i timesteps. On lines 6–7, the algorithm computes the likelihood of J_i according to the two models λ_{RC} and λ_{SD}. These likelihoods are computed using the forward algorithm. The model λ which

[1]We could consider using any arbitrary behavior-recognition algorithm, rather than the relatively straightforward HMM-based approach presented here; however, our goal here is simply to show that behavior recognition is feasible in the setting in which our team receives only incidental observations of the environment. We do not claim that the approach presented here necessarily achieves the best possible recognition accuracy in this domain.

Algorithm 6.2 Cross-validation procedure.

```
1: Given: sets of observation sequences $RC$ and $SD$, each of size $k$
2: $correct \leftarrow 0$; $incorrect \leftarrow 0$
3: for $i = 1$ to $k$ do
4:     $rc \leftarrow RC_i$
5:     $sd \leftarrow SD_i$
6:     $C_{rc} \leftarrow$ PLAY-RECOGNITION$(RC - \{rc\}, SD - \{sd\}, rc)$
7:     $C_{sd} \leftarrow$ PLAY-RECOGNITION$(RC - \{rc\}, SD - \{sd\}, sd)$
8:     $correct \leftarrow correct +$ number of elements $e$ in $C_{rc}$ s.t. $e =$ ROBOCUP
9:     $incorrect \leftarrow incorrect +$ number of elements $e$ in $C_{rc}$ s.t. $e \neq$ ROBOCUP
10:    $correct \leftarrow correct +$ number of elements $e$ in $C_{sd}$ s.t. $e =$ SUPERDEFENSE
11:    $incorrect \leftarrow incorrect +$ number of elements $e$ in $C_{sd}$ s.t. $e \neq$ SUPERDEFENSE
12: return $correct$ / ( $correct + incorrect$ )
```

maximizes the likelihood of the observation sequence is chosen as the best estimate of which play the opponent team is running. The classification output is stored in the array C.

We perform leave-one-out cross-validation to evaluate the effectiveness of our approach. Algorithm 6.2 presents the cross-validation procedure. Since we have 12 observation sequences for each case, the main loop (lines 3–12) runs 12 times. In each iteration, one element sd is held out of SD and one element rc is held out of RC. The remaining observation sequences are used to classify sd and rc, and the number of successful and unsuccessful classifications is recorded. The procedure returns the fraction of timesteps which were successfully classified.

Good classification performance requires a good domain representation, namely an informative set of *features* from the raw joint observations. In principle, we can apply any arbitrary function $f(j)$ to map a single joint observation to a vector of features. Here, we only consider feature-selection functions of a limited form. Namely, $f(j)$ can only perform two operations:

1. *Filtering* out some features, removing them from the joint observation entirely. Some of the fields communicated by the robots may not be useful for play recognition, in which case the models may overfit if these fields are present. Filtering allows useless features to be ignored.

2. *Re-ordering* features in the joint observation. By default, the joint observation is formed by simply concatenating the observation vectors of each individual robot. However, we consider re-ordering features in the joint observation so that specific positions in the joint observation vector have a semantic meaning. For example, the center of the field is $(0, 0)$ in our coordinate system and positive x points towards our own goal. Sorting our

robots' positions by x in ascending order therefore orders the positions from "nearest the opponent goal" to "furthest from the opponent goal." After this reordering, a specific index k in the transformed feature vector semantically corresponds to the x-position of our robot closest to the opponent goal, which may a more informative feature for activity recognition than "the x-position of robot i." Feature re-ordering allows such semantics to be an explicit part of the model.

To help understand which features are most important for successful play recognition in robot soccer, we tried a variety of feature-selection functions and computed the overall classification accuracy of each. These results are presented in the next section.

6.1.3 Experimental Results

By itself, ball position is the most informative feature. If the observation vectors include only the ball's x- and y-coordinates plus the "ball valid" flag, and the ball observations are sorted by their x-coordinates, the activity recognition algorithm achieves an overall classification accuracy of 86.98%.

Using only the reported x-positions of our own robots (sorted by x) achieves an accuracy of 82.34%. It is interesting to note that adding our team's own y-positions actually lowers our classification accuracy to 76.65%, due to overfitting. Adding the "own position valid" flag also hurts classification accuracy; we believe that this is because the robots' position confidence estimates were tuned for single-robot behavior, such as deciding when the robot needs to look at localization landmarks, rather than being tuned for team behavior or opponent recognition. By itself, using the x-positions of opponent robots (sorted, plus the "opponent valid" flag) performs quite poorly, achieving an accuracy of only 69.54%. Adding the y-positions of opponents also decreases classification accuracy, to 63.74%, which we also attribute to overfitting.

If the feature vector incorporates the ball position plus our team's positions, overall accuracy increases to 87.44%. Including the opponent positions as well results in the best accuracy: 88.63%. We conclude that the position of the ball is by far the most important feature; however, including additional information can help somewhat in classification performance. The importance of the ball is unsurprising given the *incidental* nature of our classification task—since most of the robots on our team spend most of their time focusing their attention on the ball, the ball is the object that is seen most and has the lowest observation error. The accuracy levels achieved with different sets of features are summarized in Figure 6.3.

Features used	Classification accuracy
$\langle$ ball x, ball y, ball valid $\rangle$	86.98%
$\langle$ own x $\rangle$	82.34%
$\langle$ opp. x, opp. valid $\rangle$	69.54%
$\langle$ ball x, ball y, ball valid, own x $\rangle$	87.44%
$\langle$ ball x, ball y, ball valid, own x, opp. x, opp. valid $\rangle$	88.63%

Figure 6.3: Summary of classification accuracy when using different sets of features as input to the HMM.

Through the selection of the proper set of features, the HMM play recognizer achieves classification accuracy of 88.63%. One question remains: in which circumstances does the classifier still make errors? Figure 6.4 shows the classification accuracy at each time step of a typical RoboCup game and a typical SuperDefense game. This figure shows that all of the classification errors occur early in the game, before the algorithm has collected many observations. After about 100 seconds have elapsed, the classification accuracy for both games is 100%. In fact, this is the typical pattern seen in almost all the games: the classification accuracy is rather poor at the beginning, but improves significantly after 100–200 seconds have elapsed. We therefore claim that in a real robot soccer scenario, we would not want to change the behavior of our own team to adjust to the opponents until either a certain amount of time has elapsed and/or until the likelihoods of the observation sequence given each of the models have diverged significantly.

6.1.4 Summary

In this section, we discussed the problem of *incidental play recognition*, in which a team of agents or robots attempts to recognize the behavior of an opponent team while engaged in a primary task (such as robot soccer) that is not optimized for behavior recognition. Our team members periodically communicate their observations of their own position, the position of the ball, and the position of opponent robots. Using an approach based on hidden Markov models, our team can accurately classify the behavior of the opponent team 88.63% of the time. Many of the classification errors are made at the very beginning of each game, when there is less data available to the classifier.

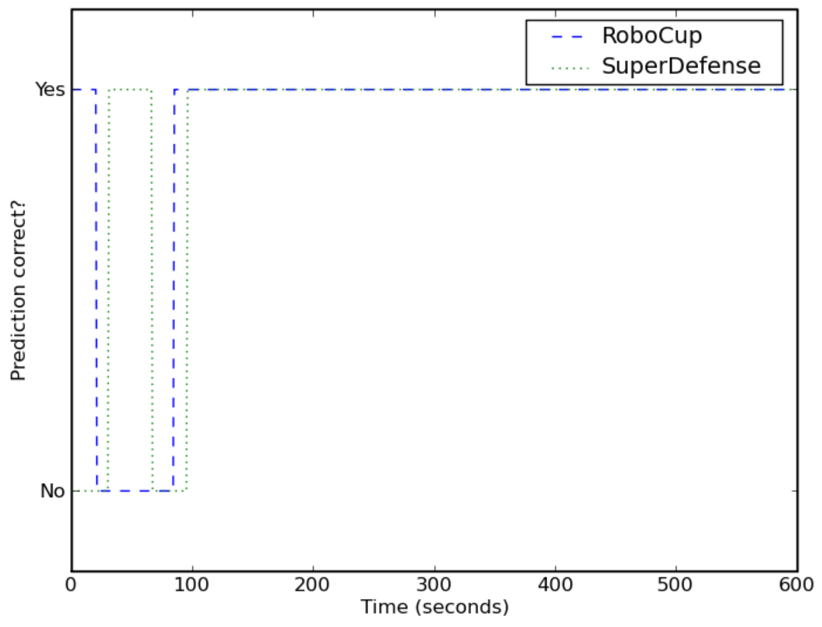


Figure 6.4: Classification accuracy at each timestep for a typical RoboCup game and a typical SuperDefense game.

6.2 Acting in Response to an Unknown Opponent

In this section, we direct our attention to the problem of playing against an opponent whose behavior is initially unknown. We consider two scenarios:

- **Static opponent play.** This scenario is equivalent to the robot soccer results presented in the previous section: the opponent is playing a fixed, but unknown, play throughout the entire game. Our team needs to recognize the play chosen by the opponents such that it can play the optimal TRSMDP policy in response.

- **Dynamic opponent play.** In this scenario, the opponent is dynamic—changing play selection over the course of the game. As in the static-opponent scenario, the opponent's play selection is initially unknown to our team. In addition, the opponent changes plays at c random times during the game. By varying c, we measure the effect of playing against opponents that change plays frequently or infrequently.

We assume the existence of an activity recognizer that has properties similar to the HMM activity recognition algorithm presented in the previous section. Namely, we assume that the activity recognizer has some *recognition delay*: an amount of time needed in order to successfully recognize the behavior of the opponent. At the beginning of the game, or whenever the opponent's play switches, the activity recognizer provides no useful data until the recognition delay has elapsed.[2] After the recognition delay has elapsed, we assume that the activity recognizer produces perfect recognition results, identifying the opponent play correctly 100% of the time. We use this idealized activity recognizer so that we can directly measure the effect of varying the recognition delay.

6.2.1 Static Opponent

We first consider the static-opponent scenario. In this scenario, the opponent is playing a single play throughout the entire game, but our team does not know which play the opponent is playing until the recognition delay d elapses. During this initial period of uncertainty, our team plays a *default play* which is intended to be effective against all possible opponents. After d time steps have elapsed, our team switches to the optimal policy against the (now known) opponent, as presented in Section 5.3.3.

We first introduced the concept of a default play in the context of robot soccer, as a failsafe coordination plan in the event of networking failures in a robot team (see Appendix A and [38, 39]). Here, we extend the concept of the default play, utilizing the default play as a safe response to an unknown opponent.

Default Play Selection

The first question we answer is: which play should our team use as the default play?[3] We find the best default play empirically, playing 3000 full games with every possible choice of our default play p_x, the opponent's static play p_y, and with the recognition delay d ranging from 200 to 1800, in increments of 200. We empirically measure the value $V_{p_x,p_y} = P(\text{winning}) - P(\text{losing})$ for each case. We apply a standard minimax reasoning procedure: we pessimistically assume that the opponent's play will be the one that minimizes V given

[2] By "provides no useful data," we mean that the activity recognizer considers all possible opponent plays to be equally probable.

[3] In general, our team might consider choosing a stochastic default play; however, we assume here that the opponent is not attempting to change plays in response to our team's play choice, so a fixed default play should be sufficient.

our choice p_x; given this, we desire to maximize V. Formally, we wish to find:

$$\operatorname*{argmax}_{p_x} \left(\min_{p_y} V_{p_x,p_y} \right) \tag{6.2}$$

Table 6.1 shows the best default play to choose in response to each opponent play in CTF, for $d = 1000$. Each row of the table shows one opponent play, the default play which performs best against that opponent play, and the expected value our team would achieve, as measured by our experiments. For $d = 1000$, the opponent play which minimizes our expected value is `A1_M1_D3`. Given the opponent's choice of `A1_M1_D3`, our team's expected value is maximized if we use `A2_M1_D2` as the default play against `A1_M1_D3`. Therefore, when $d = 1000$, the best default play is `A2_M1_D2`. In fact, it turns out that the best default play is `A2_M1_D2` for every choice of the recognition delay d.

Opponent play	Best default play	Value
`A0_M1_D4`	`A5_M0_D0`	0.8237
`A1_M1_D3`	`A2_M1_D2`	0.0257
`A2_M1_D2`	`A2_M1_D2`	0.0440
`A3_M1_D1`	`A2_M1_D2`	0.4270
`A4_M1_D0`	`A2_M1_D2`	0.5170
`A5_M0_D0`	`A4_M1_D0`	1.0000

Table 6.1: The best default play to choose in response to each opponent play for $d = 1000$, and the expected value our team would achieve as measured by our experiments.

Effect of Recognition Delay

The next question we answer is: how does the expected value change as the recognition delay d increases? We expect that the value goes down as d increases, because our team remains unaware of the opponent's play longer and is unable to exploit the optimal policy against the specific opponent until later in the game. Table 6.2 shows the empirically measured value of using `A2_M1_D2` as the default play against each of the six opponents and with d ranging from 0–1800 in increments of 200. Note that $d = 0$ corresponds to knowing the opponent's play choice immediately at the beginning of the game; the values for the $d = 0$ column are taken from Table 5.4 in Section 5.3.4.

The general trend is as expected: as d increases, the value against each play generally decreases. However, random chance in experimental results makes it hard to notice the general trend directly from the data in Table 6.2. Table 6.3 shows the same data in a different fashion: as differences in value between the known-opponent case and the unknown-opponent case. Positive entries in the table indicate values that are greater than the known-opponent case; negative entries in the table indicate values that are less than the known-opponent case. As expected, most of the entries in the table are negative (or very slightly positive, due to random chance). The bottom row of the table shows the mean value lost over all six opponent plays, for each setting of d. When d ranges from 200–600, we see that the mean value lost is very small; less than 0.01 in each case. For d from 800–1200, the mean value lost is between 0.01 and 0.02. For d of 1400 and higher, the mean value lost rises significantly: from 0.0357 to 0.0587. The fact that the loss increases rapidly as d approaches the length of the entire game reinforces our results (from Figure 4.5 in Section 4.4) that acting suboptimally near the end of the time horizon causes a much greater loss than acting suboptimally near the beginning of the time horizon.

Opp. play	$d = 0$	$d = 200$	$d = 400$	$d = 600$	$d = 800$	$d = 1000$	$d = 1200$	$d = 1400$	$d = 1600$	$d = 1800$
`A0_M1_D4`	0.8217	0.8113	0.8153	0.8167	0.8057	0.7880	0.7660	0.7523	0.7173	0.6083
`A1_M1_D3`	0.0533	0.0343	0.0440	0.0593	-0.0017	0.0257	0.0120	0.0113	0.0050	-0.0073
`A2_M1_D2`	0.0797	0.0813	0.0670	0.0640	0.0390	0.0440	0.0667	0.0127	0.0193	-0.0167
`A3_M1_D1`	0.4347	0.4360	0.4147	0.4240	0.4427	0.4270	0.4400	0.4023	0.4093	0.4227
`A4_M1_D0`	0.5227	0.5417	0.5213	0.5227	0.5197	0.5170	0.5403	0.5193	0.5287	0.5527
`A5_M0_D0`	0.9993	0.9997	1.0000	1.0000	0.9997	0.9997	1.0000	0.9993	0.9993	0.9993

Table 6.2: The empirically measured value of using `A2_M1_D2` as the default play against each of the six opponents, for each value of the recognition delay d.

Opp. play	$d = 200$	$d = 400$	$d = 600$	$d = 800$	$d = 1000$	$d = 1200$	$d = 1400$	$d = 1600$	$d = 1800$
`A0_M1_D4`	-0.0104	-0.0064	-0.0050	-0.0160	-0.0337	-0.0557	-0.0694	-0.1044	-0.2134
`A1_M1_D3`	-0.0190	-0.0093	0.0060	-0.0550	-0.0276	-0.0413	-0.0420	-0.0483	-0.0606
`A2_M1_D2`	0.0016	-0.0127	-0.0157	-0.0407	-0.0357	-0.0130	-0.0670	-0.0604	-0.0964
`A3_M1_D1`	0.0013	-0.0200	-0.0107	0.0080	-0.0077	0.0053	-0.0324	-0.0254	-0.0190
`A4_M1_D0`	0.0190	-0.0014	-0.0000	-0.0030	-0.0057	0.0176	-0.0034	0.0060	0.0300
`A5_M0_D0`	0.0004	0.0007	0.0007	0.0004	0.0004	0.0007	0.0000	0.0000	-0.0000
Mean Loss	-0.0012	-0.0082	-0.0041	-0.0177	-0.0183	-0.0144	-0.0357	-0.0387	-0.0587

Table 6.3: Difference between the empirically measured values of the known-opponent case and using `A2_M1_D2` as the default play against each of the six opponents, for each value of the recognition delay d.

6.2.2 Dynamic Opponent

We consider the dynamic-opponent scenario, in which the opponent changes plays over the course of the game. As in the static-opponent scenario, the opponent's play selection is initially unknown to our team. In addition, the opponent changes plays at c random times during the game. By varying c, we measure the effect of playing against opponents that change plays frequently or infrequently.

We restrict the opponent's play choice here to the two plays that perform best against the thresholded-rewards CTF policies: `A1_M1_D3` and `A2_M1_D2`. In Section 5.3.4, we present results showing that the empirically-measured values of the optimal policies against these two opponent plays are 0.0533 and 0.0797, respectively. The opponent's initial play choice is chosen at random between `A1_M1_D3` and `A2_M1_D2`. Before the game begins, c times are chosen uniformly at random between 0 and 2000; at these times, the opponent team will switch plays. By varying c, we measure the effect of playing against opponents that switch plays frequently or infrequently.

We set the recognition delay d to 200 for all the experiments performed in this section.[4] The opponent's initial play choice is unknown to our team for the first 200 time steps; furthermore, the opponent's play choice becomes unknown for the next 200 time steps whenever the opponent switches plays. As in the previous section, our team's policy is to choose `A2_M1_D2` as the default play when the opponent's play is unknown; when the opponent's play is known, our team plays the optimal policy in response, as presented in Section 5.3.3.

We performed a set of experiments, varying the number of times the opponents change their play, c, from 1 to 10. For each condition, 3000 CTF games were played. Table 6.4 presents our results. Each row of the table shows the number of wins, losses, and ties achieved by our team for each value of c, and the empirically measured value of each condition: (# wins $-$ # losses) / 3000. We were initially surprised to see that our team's expected value actually increases as c increases: from 0.0480 at $c = 1$ to 0.1007 at $c = 10$. As in Section 5.3.4, we attribute this to the cost of switching plays: every time the opponent team changes plays, it takes some time for the team members to reconfigure themselves into their new roles, and during this transition time the opponents are vulnerable to having the flag taken. It turns out that the opponent team's cost of switching plays is greater than our loss due to not playing the optimal strategy against the opponent, so our expected value increases as the opponent switches plays more frequently.

To get a better idea of how much value our team does lose due to the effect of not knowing

[4]In the CTF domain, 200 time steps is 10% of the total game length; this is roughly the same proportion of the game as the recognition delay experienced by our real robot soccer team.

c	Wins	Losses	Ties	Value
1	1057	913	1030	0.0480
2	1089	969	942	0.0400
3	1096	922	982	0.0580
4	1109	905	986	0.0680
5	1129	916	955	0.0710
6	1137	904	959	0.0777
7	1149	910	941	0.0797
8	1161	905	934	0.0853
9	1185	918	897	0.0890
10	1198	896	906	0.1007

Table 6.4: Results of playing against a dynamic opponent that switches between `A1_M1_D3` and `A2_M1_D2` at c random times throughout the game. Each row of the table shows the number of wins, losses, and ties achieved by our team for each value of c, and the empirically measured value of each condition.

the opponent's play choice, we performed an additional experiment in which the opponent's cost of switching plays is reduced significantly. This experiment was identical to the previous experiment, but whenever the opponents switched plays, any opposing players in the Defender role were immediately teleported to their desired positions near the flag, significantly reducing the cost of switching plays for the opponent. However, this teleportation effect was only applied to the opposing team; our team still experienced the normal cost of switching plays. The opponents' ability to teleport thus presents the worst-case scenario for our team, as we cannot take advantage of the opponent team when they switch plays, but the opponents can take advantage of our team when we switch plays. However, since our team's abilities remain unchanged, the results of this experiment are directly comparable to the results presented earlier in this chapter and in Chapter 5.

We performed another set of experiments, varying c from 1 to 10. For each condition, 6000 CTF games are played. Table 6.5 and Figure 6.5 present our results. Each row of Table 6.5 shows the number of wins, losses, and ties achieved by our team for each value of c, and the empirically measured value of each condition: (# wins $-$ # losses) / 6000. Figure 6.5 shows the same data, plotted with the number of opponent play switches per game on the x-axis and the empirically measured value on the y-axis. As c increases from $c = 1$ to $c = 4$, we observe that the empirical value falls from 0.0443 to near zero. As c increases, the opponent switches plays more frequently, and so our team is unaware of the opponent's play for larger proportions of the game. Our team is therefore forced to play suboptimally more often, choosing the default play instead of the optimal thresholded-rewards policy. However,

c	Wins	Losses	Ties	Value
1	2067	1801	2132	0.0443
2	2059	1850	2091	0.0348
3	2050	1960	1990	0.0150
4	1999	2004	1997	-0.0008
5	2048	2007	1945	0.0068
6	2022	2015	1963	0.0012
7	2050	2004	1946	0.0077
8	2051	2012	1937	0.0065
9	2049	2082	1869	-0.0055
10	2078	2121	1801	-0.0072

Table 6.5: Results of playing against a dynamic opponent that switches between `A1_M1_D3` and `A2_M1_D2` at c random times throughout the game. Each row of the table shows the number of wins, losses, and ties achieved by our team for each value of c, and the empirically measured value of each condition.

from $c = 4$ to $c = 10$, the value remains essentially unchanged, falling within $[-0.01, 0.01]$ in all cases. For $c \geq 4$, our team no longer knows the opposing team's play often enough to effectively play the optimal thresholded-rewards strategy in response. However, it is reassuring to note that our performance does not fall significantly below 0. This indicates that, even in the worst case, our choice of a safe default play ensures that our team has the same probability of winning as the opponent.

6.3 Summary

In this chapter, we relaxed the assumption that the play choice of the opponent is static and known to our team *a priori*. We introduced the problem of *incidental behavior recognition*, in which our team has some primary task beyond simply classifying the behavior of other agents operating in the same environment. Experiments with our robot soccer team demonstrate that, with the right set of state features, our team can accurately classify the behavior of the opponent team 88.63% of the time. Many of the classification errors are made at the very beginning of each game, when there is less data available to the classifier.

In the CTF domain, we considered a *static opponent* scenario in which the opponent is playing a fixed, but initially unknown, play throughout the entire game and a *dynamic opponent* scenario in which the opponent changes plays over the course of the game. We

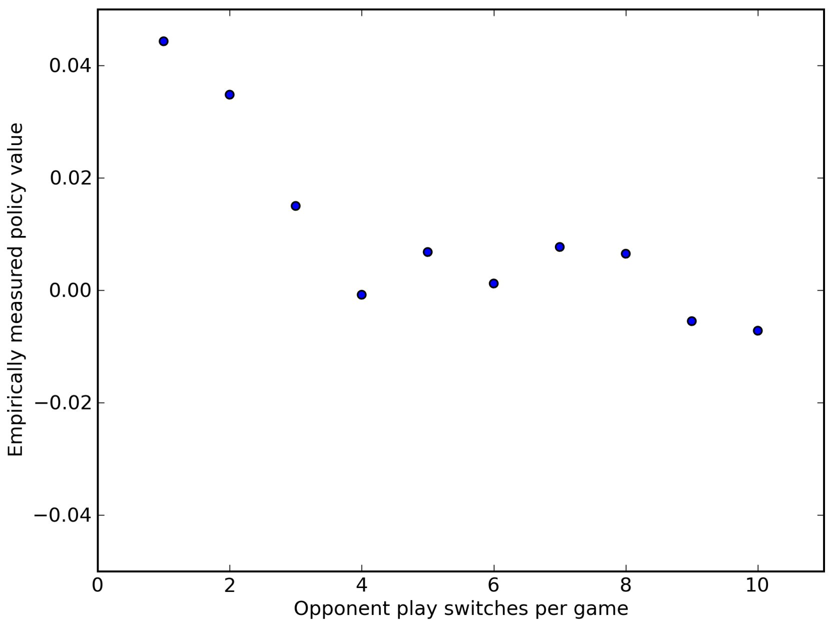


Figure 6.5: Results of playing against a dynamic opponent that switches between `A1_M1_D3` and `A2_M1_D2` at c random times throughout the game. The x-axis shows the number of opponent play switches per game; the y-axis shows the empirically measured value.

extended the concept of a *default play*, originally introduced as a failsafe coordination plan in the event of network failures in a robot team, as a safe response to an unknown opponent.

Our experiments in the static-opponent case show that `A2_M1_D2` is the best default play to use in the CTF domain. We also performed experiments that show how our expected value is dependent on the *recognition delay*: the amount of time it takes our team to successfully recognize the behavior of the opponent team. As the recognition delay increases, our expected value decreases.

In the dynamic-opponent case, we found that the opponent's cost of play switching is greater than our team's cost of not knowing which play the opponent is playing; therefore our team performs better against an opponent which switches plays more frequently. We performed a further experiment in which the opponent's cost of switching plays was artificially reduced, by teleporting the opponent's defenders into their desired positions immediately upon initation of the new play. In this scenario, our team performs worse when the opponent switches plays more frequently; however, our value never falls significantly below 0, indicating that our choice of a safe default play ensures that our team has the same probability of winning as the opponent.

The results presented in this chapter show that it is possible for our real robot soccer team to effectively recognize the behavior of the opponent team, even though our observations of the environment are incidental. Furthermore, we have showed that the combination of a default play with the optimal thresholded-rewards policy can lead to effective performance in situations in which the behavior of the opponent team is initially unknown, and in which the opponent switches plays during execution.

Chapter 7

TRMDPs with Unknown Rewards

In this chapter, we focus on a control problem inspired by reCAPTCHA: given a document containing w unknown words and a hard time deadline h, how can the challenge generator choose challenges in order to maximize the probability of successfully transcribing the entire document on time? (Section 2.3 presents a full description of the reCAPTCHA domain.) This control problem is challenging because the reCAPTCHA system does not know whether an answer to an unknown word is correct. We are therefore posed with the difficult problem of trying to obtain some given amount of reward (number of words digitized) without actually knowing the amount of reward achieved during execution.

In the following section, we present a detailed model of the reCAPTCHA domain that is derived from over 29 million answers gathered by the real reCAPTCHA system over a six-month period. In Section 7.2, we briefly present a summary of the algorithms from previous chapters that we use to tackle the challenge of unknown rewards. In Section 7.3, we present an algorithm that aims to address the problem of unknown rewards by taking periodic samples of reward values. In Section 7.4, we experimentally evaluate the performance of this algorithm against the optimal thresholded-rewards policy, the policies generated by the *uniform-k* heuristic, and the policy that maximizes expected rewards (without considering time or cumulative reward). We also compare the effects of two different reward estimation functions.

7.1 reCAPTCHA Domain Model

We model the reCAPTCHA domain as an MDP (S, A, T, R). Each step in the MDP corresponds to a single user requesting a CAPTCHA challenge. The state of the system represents whether the reCAPTCHA system is currently "under attack" by a large number of malicious users. Given the state, the challenge generator then needs to choose an action—which type of CAPTCHA challenge to send to the user. We consider three types of challenges: *standard*, in which the user is shown one known word and one unknown word; *two-known*, in which the user is shown two known words; and *two-unknown*, in which the user is shown two unknown words. Reward is gained when a user provides a correct solution to an unknown word and lost when a user provides an incorrect solution to an unknown word.

The reward distribution for the reCAPTCHA domain is derived by analyzing the answers provided by 31,163 of the most active reCAPTCHA users.[1] Over a six-month period, these users submitted over 29 million answers to reCAPTCHA. We measured each user's solution accuracy; Figure 7.1 shows a histogram of per-user solution accuracies. Based on this histogram, we have partitioned the users into three classes:

1. *Accurate users.* This group consists of the 27,286 users that attain a solution accuracy of 85% or higher. These users submitted a total of 28,921,000 answers; 95.22% of these answers were correct. Accurate users constitute the majority of reCAPTCHA users.
2. *Inaccurate users.* This group consists of the 3,807 users with solution accuracies in the range [10%-85%). These users submitted a total of 873,000 answers; 66.88% of these answers were correct. Inaccurate users are not necessarily trying to seed incorrect answers into the system; many of them are from countries where English is not a commonly-spoken language. This makes it more difficult to pass the reCAPTCHA challenge, since most of the reCAPTCHA challenges consist of English words.
3. *Malicious users.* This group consists of 70 users with solution accuracies lower than 10%. These users submitted a total of 108,000 answers; 0.44% of these answers were correct. Though these users provide a small proportion of total answers, it is important to consider them in our analysis because their traffic can be "bursty"; i.e. multiple malicious users may simultaneously submit many bad answers to reCAPTCHA as part of an organized attack. After time passes, the malicious users generally disappear.

[1]In this analysis, we assume that each unique Internet Protocol (IP) address corresponds to a single user. In reality, due to network address translation (NAT) and proxies, one IP address may be shared by multiple humans. Conversely, a single attacker may use a network of multiple machines (a "botnet") to submit answers from a large number of IP addresses. However, these distinctions are mostly irrelevant for the analysis presented here.

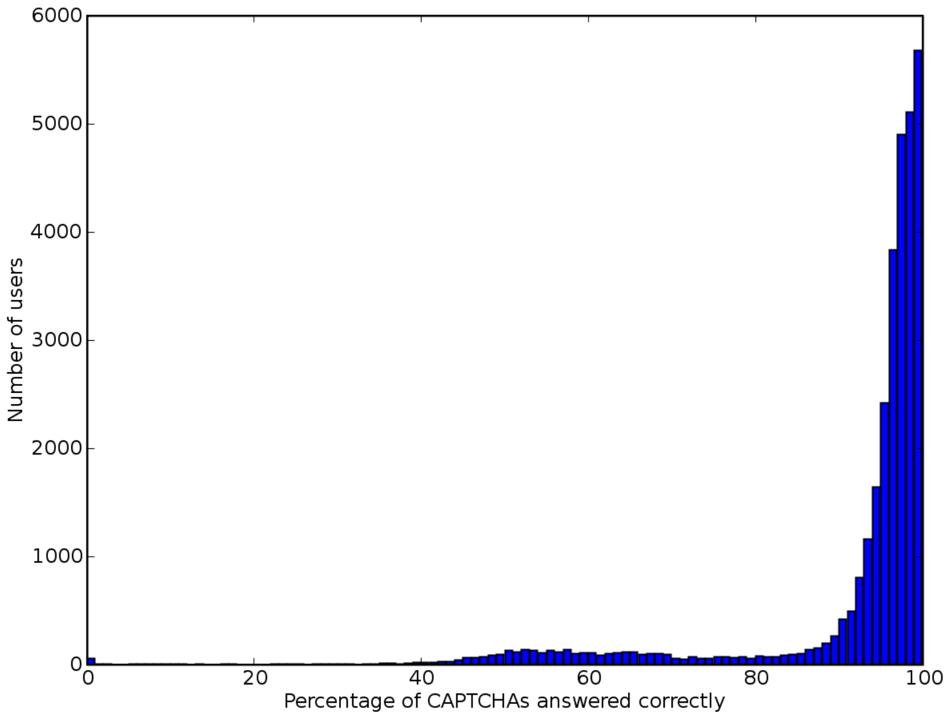


Figure 7.1: Histogram of per-user solution accuracy for 31,163 of the most active reCAPTCHA users.

Figure 7.2 shows the three states of our MDP model: *accurate*, *mixed*, and *attack*. The *accurate* state corresponds to the most common case, when nearly all the users are "accurate": answering reCAPTCHA challenges with high accuracy. In the *mixed* state, we assume that half the users are "accurate" and half the users are "inaccurate." We therefore expect that a standard reCAPTCHA challenge will get answered correctly approximately 81% of the time. In the *attack* state, we similarly assume that half the users are "accurate" and half the users are "malicious," leading to an overall accuracy rate of approximately 48%.

Figure 7.3 shows the reward distribution for our model, which is derived from actual reCAPTCHA answer data. We assume that we get 1 point of reward for every unknown word answered correctly by a user, and -2 points of reward for every word answered incorrectly, as it generally takes two correct answers to override each incorrect answer when it comes time to produce the final output. For a *standard* CAPTCHA challenge, the agent receives reward of either 1 or -2. For a *two-unknown* challenge, the agent receives reward of 2 (if both words are answered correctly), -1 (if one word is answered correctly and one word is answered incorrectly), or -4 (if both words are answered incorrectly). For a *two-known* challenge, no rewards are obtained because no unknown words are presented to the user.

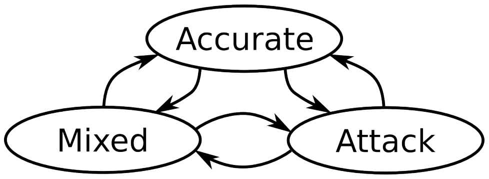


Figure 7.2: MDP model of the reCAPTCHA domain.

$R(s,a)$	$a =$ *standard*	$a =$ *two-unknown*	$a =$ *two-known*
$s =$ *accurate*	1 (p=0.9522) -2 (p=0.0478)	2 (p=0.7067) -1 (p=0.1910) -4 (p=0.1023)	0 (p=1.0)
$s =$ *mixed*	1 (p=0.8105) -2 (p=0.1895)	2 (p=0.5569) -1 (p=0.3572) -4 (p=0.0859)	0 (p=1.0)
$s =$ *attack*	1 (p=0.4783) -2 (p=0.5217)	2 (p=0.1288) -1 (p=0.5490) -4 (p=0.3222)	0 (p=1.0)

Figure 7.3: Reward distribution for the reCAPTCHA domain.

Figure 7.4 shows the transition probabilities for our domain, which model the fact that the most common state is *accurate*, followed by *mixed*, with *attack* occurring only a small proportion of the time. We assume that our choice of action does not affect the transition probabilities; therefore the transition probabilities only depend on the current state.

In summary, our model of the reCAPTCHA domain is an MDP $M = (S, A, T, R)$, where $S = \{$ *accurate*, *mixed*, *attack* $\}$, $A = \{$ *standard*, *two-unknown*, *two-known* $\}$, T is given by Figure 7.4, and R is 1 for every unknown word answered correctly and -2 for every unknown word answered incorrectly (given by Figure 7.3).

$T(s,*,s')$	$s' =$*accurate*	$s' =$*mixed*	$s' =$*attack*
$s =$*accurate*	0.9	0.09	0.01
$s =$*mixed*	0.09	0.9	0.01
$s =$*attack*	0.09	0.01	0.9

Figure 7.4: Transition probabilities for the reCAPTCHA domain.

7.2 Background

In previous chapters, we have focused on the problem of timed, zero-sum games such as robot soccer and Capture the Flag, in which the goal is to be ahead at the end of the time horizon. In this chapter, we address a domain that also has a hard time deadline, but is not zero-sum. Instead, our "score" is the number of words successfully digitized so far. We model the reCAPTCHA domain as an MDP, as described above; the threshold function f is the *zero-one reward threshold function* introduced in Section 3.1:

$$r_{true} = \begin{cases} 1 & \text{if } r_{intermediate} \geq w \\ 0 & \text{otherwise.} \end{cases} \tag{7.1}$$

Recall from Chapter 3 that the optimal policy for a TRMDP depends on the state, the time remaining, and the cumulative reward actually obtained during execution time. However, in the reCAPTCHA domain, we do not know the cumulative reward—since reCAPTCHA does not know the correct spelling of each word, the system does not know how much cumulative reward has been obtained during execution time. In this chapter, we extend the TRMDP approach to domains with *unknown rewards* by presenting a sampling-based algorithm that chooses actions based on an estimate of the cumulative reward achieved so far.

In Chapter 4, we presented heuristic solution methods for TRMDPs. One of these heuristic approaches is *uniform-k* (Section 4.1), in which the policy only considers changing actions every k steps, in order to save computation time. In this chapter, we find that the *uniform-k* heuristic is also useful for the development of our sampling-based approach.

It is important to note that the problem of unknown rewards in a thresholded-rewards domain is distinct from the problem of an unknown or unmodeled domain, such as is addressed by reinforcement learning [22]. In reinforcement learning, the agent does not know the transition or reward functions *a priori*, but must learn about the domain using the transitions and rewards experienced during execution. Rather, we assume that the transition and reward functions are known, but the actual reward values received during execution are unknown.

7.3 Sampling-Based Control Policy

The optimal policy in a thresholded-rewards domain depends on the time remaining and the cumulative reward obtained by the agent. In this section, we present our sampling-based control policy which aims to address the challenge of having unknown rewards at execution

Algorithm 7.1 Sampling-based control policy.

```
1: Given: MDP $M$, threshold function $f$, time horizon $h$, sampling interval $k$, reward
   estimation function $E$
2: $\pi \leftarrow$ Uniform$(M, f, h, k)$
3: $S \leftarrow \{\}$        // set of reward samples
4: $s \leftarrow s_0$        // current state
5: for $t \leftarrow h$ to 1 do
6:     if $|S| = 0$ then
7:         $\hat{r} \leftarrow 0$
8:     else
9:         $\hat{r} \leftarrow$ nearest_integer$(E(S))$
10:    $a \leftarrow \pi(s, t, \hat{r})$
11:    $(s, r) \leftarrow$ Act$(M, a)$
12:    when $r$ then    // get reward sample (every $k$ steps)
13:        $|S| \leftarrow |S| \cup \{r\}$
```

time. We assume that our agent occasionally observes the reward received when taking an action; for the reCAPTCHA domain, we assume we have access to an *oracle*: a trusted human who checks the results of every kth CAPTCHA response and verifies whether the solutions to the unknown words (if any) were correct.[2] The agent will only consider changing its policy when it receives a reward sample, which means that the agent's policy is equivalent to the *uniform-k* policy except that we use an estimate of the cumulative reward rather than the true cumulative reward.

Algorithm 7.1 shows our sampling-based control policy for thresholded-rewards domains with unknown rewards. This algorithm takes the same inputs as the TRMDP solution algorithm (Algorithm 3.2): an MDP M, threshold function f, and time horizon h. For the reCAPTCHA domain, f is the zero-one threshold function shown in Equation 7.1: we receive reward 1 if the cumulative reward is greater than or equal to the number of unknown words w in the document; 0 otherwise. Algorithm 7.1 also takes as input an integer k, the number

[2]Instead of having a trusted human who checks every kth answer, in the reCAPTCHA domain we could instead verify every kth CAPTCHA response by asking n additional users to transcribe the same word. The initial response is judged to be correct iff if a majority of the other users agree with the initial response. The risk of doing this is that we could get "tricked" if many of the users are providing malicious answers. As long as there are at least some honest users in the system, we can set n high enough that getting tricked is an event that happens with arbitrarily low probability. However, the downside to this approach is that setting a high n means that there are fewer "useful" human answers available. In this chapter, we don't consider how this oracle might be implemented; we just assume that some oracle exists, and that querying the oracle has some non-trivial cost—otherwise we would just query the oracle at every single timestep.

of time steps elapsed between each reward sample; and a reward estimation function E, which takes in a set of reward samples S and outputs an estimate of how much cumulative reward the agent has received so far. In this chapter, we consider two reward estimation functions. The first, Mean, computes the mean of the samples, then multiplies this value by the total number of steps taken so far:

$$\textsc{Mean}(S) = \frac{\sum_{r \in S} r}{|S|} \times (h - t) \tag{7.2}$$

The other reward estimation function we consider is Low, which estimates the per-step reward as the lower bound of the 95% confidence interval of the samples seen so far:

$$\textsc{Low}(S) = \text{CI-}\textsc{Lower-Bound}(S) \times (h - t) \tag{7.3}$$

Line 2 of Algorithm 7.1 first calls the *uniform-k* TRMDP solution heuristic. This returns the best policy π that only considers changing strategies every k time steps. On lines 3–4, we initialize the set of reward samples to the empty set and the current state of the system to be the initial state of MDP M. Line 5 begins the main control loop of the agent: in each iteration through this loop, the agent executes a single action. Lines 6–9 determine the reward estimate $\hat{r}$ that will be used to determine the action. If the agent has not yet received any reward samples, it assumes that the cumulative reward is 0. If the agent does have samples, it calls the reward estimation function to estimate the cumulative reward. The result of the reward estimation function is rounded to the nearest integer, because the policy π returned by the *uniform-k* solution algorithm requires integer reward values. Line 10 determines the optimal action to take given the current state, the time remaining, and our estimate of the cumulative reward. On line 11, the agent acts in the world. As a result of this action, the agent receives the new state of the system s, and may also receive knowledge of the reward r it received for taking that action. If a reward sample is received, the agent adds it to the set of reward samples seen. Regardless, the loop then continues at line 5, until h time steps have elapsed and the process completes.

7.4 Results

In this section, we present results showing the effectiveness of our algorithm in the reCAPTCHA domain. First, we consider the question: how effective would an agent be if the agent actually knew the reward it received at execution time? By running the optimal TRMDP solution algorithm on this domain, we can find the value of the optimal policy, which serves as an upper bound on the value of any solution algorithm for this domain.

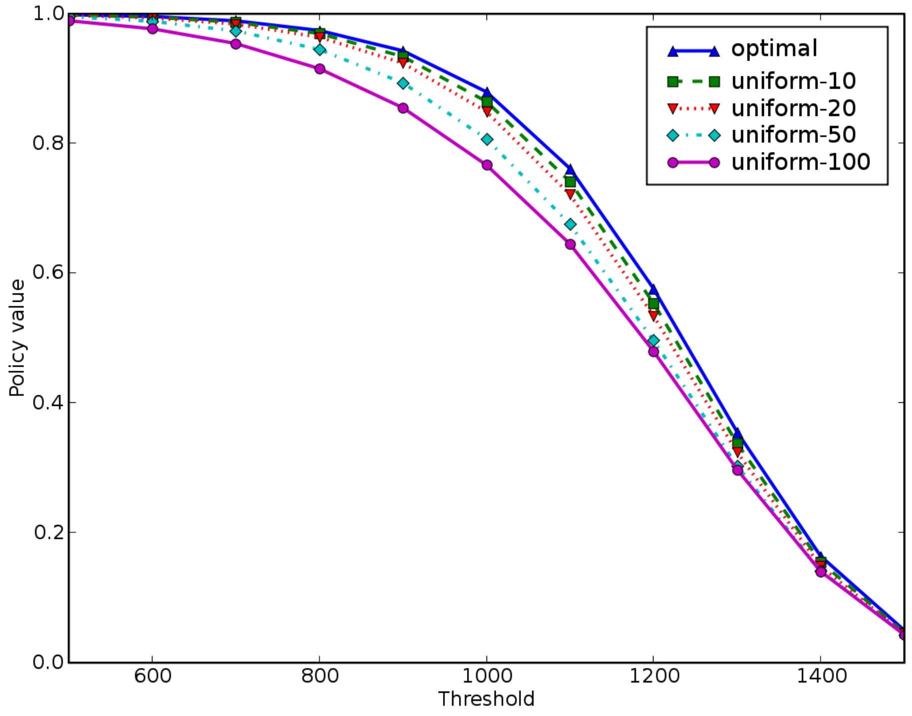


Figure 7.5: Value of the optimal policy and uniform-k for the reCAPTCHA domain, with the time horizon $h = 1000$, values of k in $\{10, 20, 50, 100\}$, and thresholds ranging from 500–1500.

Further, the *uniform-k* algorithm provides the optimal policy that only considers changing policy every k steps. This gives an upper bound on value of any policy that samples every k steps: if the reward estimation function E always returns the actual cumulative reward, the sampling policy will always choose the optimal *uniform-k* action; however, if E estimates incorrectly, the sampling policy might choose a suboptimal action.

Figure 7.5 shows the results of running the optimal solution algorithm and *uniform-k* heuristic on the reCAPTCHA domain, with the time horizon set to $h = 1000$ steps, threshold values ranging from 500 to 1500, and k set to 10, 20, 50, and 100. The value of each policy is equal to the probability that an agent following that policy will achieve the desired reward threshold. For a threshold of 500 to 800 words, the figure shows that the agent can achieve the threshold with near certainty ($> 95\%$). As the threshold increases, the probability of achieving the reward threshold decreases, but the optimal algorithm can still succeed over 50% of the time when the threshold is set to 1200. The value drops off sharply after that point; with the reward threshold set to 1500, the target is achieved less than 5% of the time. The *uniform-k* policies perform worse as k increases; this is unsurprising since higher values of

k correspond to rougher approximations to the optimal policy. However, even *uniform*-100 has performance that is reasonably close to optimal.

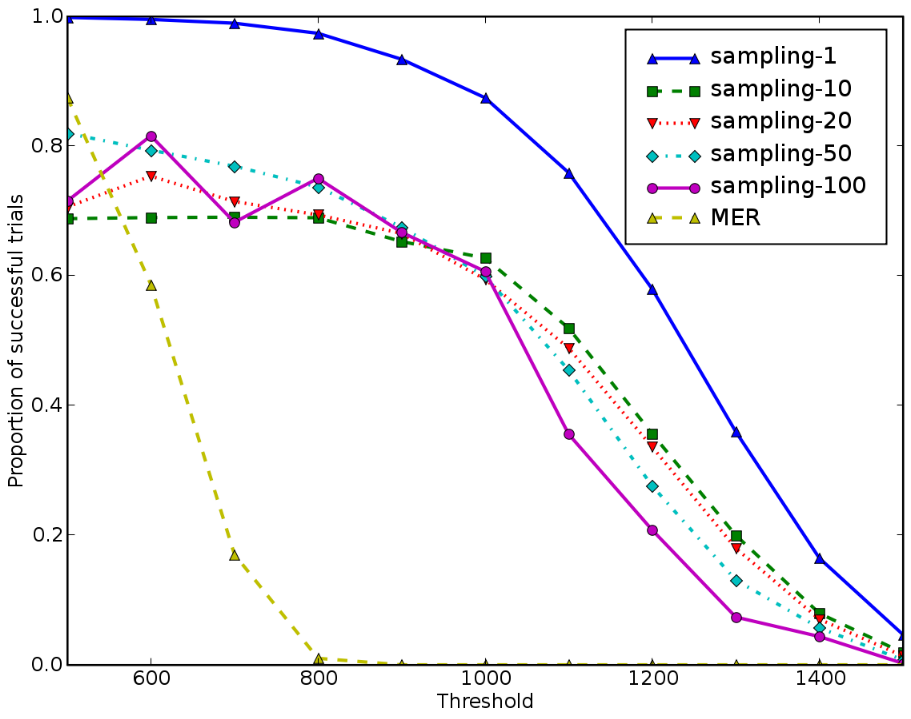


Figure 7.6: Results when sampling using the MEAN reward estimation function on the reCAPTCHA domain, with time horizon $h = 1000$, values of k in $\{1, 10, 20, 50, 100\}$, and thresholds ranging from 500–1500. The y-axis shows the proportion of 3,600 trials which were successes. The success rate of the maximize-expected-rewards ("MER") policy is also shown.

Figure 7.6 shows the results of our sampling algorithm when the MEAN function is used to estimate the reward. Again, we have set the time horizon to $h = 1000$ steps and thresholds ranging from 500–1500. For the sampling algorithm, k determines how often the agent receives a reward sample, and is set to 1, 10, 20, 50, or 100. *Sampling*-1, which receives a reward sample at every time step, is equivalent to the optimal policy. For each combination of k and threshold, we ran 3,600 trials; the graph shows the fraction of these trials which were successes (i.e. in which the reward threshold was met). For comparison with the non-thresholded-rewards solution, we also show the success rate of the policy that simply maximizes expected rewards ("MER"). The MER policy always chooses the *standard* CAPTCHA challenge when the MDP is in the *accurate* or *mixed* states, and the *two-known* challenge when the MDP is in the *attack* state. Since the MER policy does not depend on the reward received so far, it chooses an action at every time step (as does *sampling*-1).

Figure 7.6 clearly shows that the sampling-based thresholded-rewards policy outperforms the policy that maximizes expected rewards. However, there is a significant gap between the sampling-based policies and the theoretical upper bounds shown in Figure 7.5. With the Mean reward estimator, agents' reward estimates are effectively "too optimistic" in a significant number of trials. An agent that believes it has enough reward to meet the threshold will begin to act conservatively, selecting the *two-known* CAPTCHA because the *two-known* CAPTCHA cannot lead to negative reward. If the reward estimate is correct, this is indeed the best strategy, but if the reward estimate is too high, the agent's "complacent" choice of the *two-known* CAPTCHA prevents it from achieving the remaining reward that is needed. When the desired reward is relatively easy to achieve, the chance of incorrectly falling into this "complacent" strategy is higher, explaining the relatively large gap between the sampling-based policies and the theoretical upper bounds when the threshold value is low.

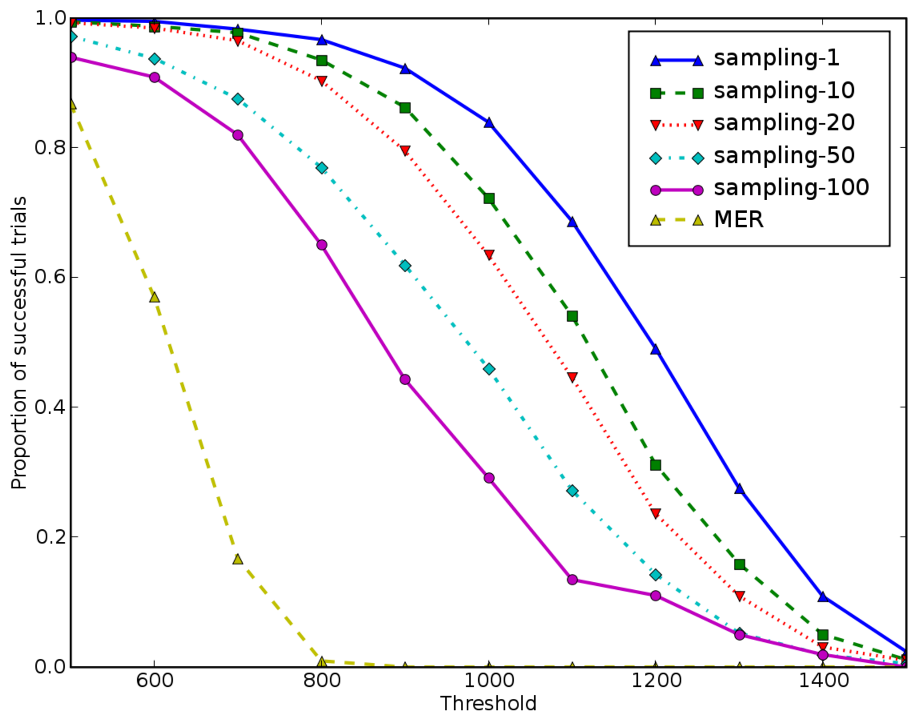


Figure 7.7: Results when sampling using the Low reward estimation function on the reCAPTCHA domain, with time horizon $h = 1000$, values of k in $\{1, 10, 20, 50, 100\}$, and thresholds ranging from 500–1500. The y-axis shows the proportion of 3,600 trials which were successes.

Figure 7.7 shows the results of our sampling algorithm when the Low function is used to estimate the reward. The Low function is more pessimistic with regard to the estimated reward; it assumes that the average per-step reward is the lower bound of the 95% confidence interval of the reward samples seen so far. Since the Low function underestimates the reward (compared to Mean), an agent using the Low function is unlikely to erroneously fall into the "complacent" strategy. The results indicate that the performance of the Low function is better than the performance of Mean for almost every setting of k and the threshold value. The performance of *sampling*-10 nearly matches the theoretical upper-bound of *uniform*-10. As expected, the overall performance degrades as we take samples less often. However, even *sampling*-100 significantly outperforms the maximize-expected-rewards policy, which is quite impressive since the *sampling*-100 agent only receives 10 reward samples over the entire time horizon.

These results show that there is a significant benefit to reasoning about reward and time in thresholded-rewards domains, even if our agent only obtains a small sample of the reward values received during execution.

7.5 Summary

In this chapter, we have presented a sampling-based algorithm that enables agents to reason about cumulative rewards and time deadlines in domains where the exact rewards achieved by the agent are not known to the agent at execution time. Our results in previous chapters have been primarily directed towards zero-sum games; in this chapter we addressed a problem that has a hard time deadline and a notion of "score", but that is not a game and does not have a specific opponent. Using the reCAPTCHA domain, we have shown that reasoning about time and "score" can lead to a significant benefit, even if we only obtain a small sample of reward values during execution time.

We have experimentally tested the effectiveness of two possible reward estimation functions: Mean, which estimates cumulative reward by using the mean of all samples seen so far; and Low, which estimates cumulative reward by using the lower bound of the 95% confidence interval. If the Mean function happens to overestimate the reward achieved by the agent, this can potentially cause the agent to adopt an overly conservative strategy, which detracts from overall performance. In comparison, the Low function provides a somewhat pessimistic estimate of the reward obtained by the agent, which prevents the agent from erroneously adopting an overly conservative strategy.

Chapter 8

Related Work

In this chapter, we discuss lines of research related to the work presented in this thesis. Section 8.1 presents previous results in the area of Markov decision processes (MDPs) and semi-Markov decision processes (SMDPs). Section 8.2 discusses decision problems with alternative objective functions, such as non-Markovian rewards and risk-sensitive utility functions. Section 8.3 presents results from the general area of multi-robot teamwork; Sections 8.4 and 8.5 present specific results from the fields of robot soccer and American football.

8.1 Markov Decision Processes

Markov Decision Processes (MDPs) are a powerful tool for planning in the presence of uncertainty [46]. MDPs provide a theoretically sound means of achieving optimal rewards in uncertain domains. Since we interested in domains with stochastic actions, we make significant use of MDPs in this thesis.

An MDP is represented as a tuple (S, A, T, R, s_0). S is a set of states and A is a set of actions. s_0 is the initial state. T is the transition function; $T(s, a, s') \rightarrow [0, 1]$ denotes the probability of transitioning to state s' when action a is executed in state s. R is the reward function; $R(s, a)$ denotes the reward received upon taking action a when in state s. Alternatively, rewards can be found on the states, in which case the agent receives reward $R(s)$ upon entering state s. A *policy* is a mapping $\pi : S \times \mathcal{N} \rightarrow A$ from $\langle$state, time$\rangle$ pairs to actions. A *stationary policy* is a policy which does not depend on time; a policy that does depend on time is called *nonstationary* or *time-dependent* [42].

The typical goal of decision making in an MDP is to find an optimal policy. In order to do this, an *optimality criterion* (also known as an *objective function*) needs to be defined. Many optimality criteria for MDPs have been proposed in the literature (see [34, 35, 46] for extensive discussion). The most commonly used optimality criteria include:

- Expected cumulative reward
- Expected cumulative discounted reward
- Average reward rate (expected reward per time step).

In the *expected cumulative reward* and *expected cumulative discounted reward* frameworks, the *value* of state s under policy π is denoted as:

$$V_\pi^k(s) = E\Big[\sum_{t=0}^{k} \gamma^t r_t\Big], \tag{8.1}$$

where r_t is the reward received at time t, $\gamma \in (0, 1]$ is a discount factor applied to future rewards, and k is the *horizon*: the number of time steps until completion. When $k = \infty$, we have an *infinite-horizon problem*; when k is finite, we have a *finite-horizon problem.* The discount factor γ is generally used for the formulation of infinite-horizon problems; otherwise, the cumulative reward is likely to grow without bound. For finite-horizon problems, such as those addressed in this thesis, infinite rewards are not possible, so $\gamma = 1$ is typically used.

A common goal of decision making in an MDP is to find a policy π that maximizes $V_\pi^k(s)$ for every $s \in S$. Such a policy is said to be *optimal.* The optimal value $V^k(s)$ of s is the expected discounted future reward received when we start in state s and follow an optimal policy. V^k can be computed exactly using dynamic programming techniques; this process is known as *value iteration* [4]. Value iteration uses the Bellman equation, which is:

$$V^{k+1}(s) = \max_{a \in A} \Big\{ R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V^k(s') \Big\}, \tag{8.2}$$

where $V^0(s) = 0$. For the finite-horizon case with k timesteps remaining, V^k allows us to easily find the optimal next action from any state. However, the optimal action from a given state may depend on the number of time steps remaining; such a policy is said to be *nonstationary.* For the infinite-horizon case, V^k converges to some V^* as $k \to \infty$ (for $\gamma < 1$). A related method of computing the optimal policy is *policy iteration* [19], in which an initial policy π is iteratively improved using the value function V_π until it converges to the optimal policy π^* (with value V^*).

MDPs assume that the agent is provided with an appropriate model of the environment *a priori*; if this is not the case, the agent needs to learn about the environment online. *Reinforcement learning* is one common method for doing so [22, 68, 79]. In *model-free* reinforcement learning, the agent learns a policy without learning a model of the environment; in *model-based* reinforcement learning, the agent learns a model of the environment and uses this model to derive a policy. In this thesis, we do not directly address the issues of online reinforcement learning in thresholded-rewards domains. However, in Chapters 5 and 6 our models of the robot soccer and CTF domains are gathered empirically in an offline fashion. This offline step finds an estimated transition function $\hat{T}$ in a similar fashion to online model-based reinforcement learning methods [41, 67]. Interval estimation has been proposed as an effective means of trading off exploration and exploitation in reinforcement learning [23, 62, 63, 81]. We make use of an interval estimation approach in Chapter 7, utilizing the Low reward estimation function (Equation 7.3) to provide a conservative estimate of the cumulative intermediate reward received during execution.

In this thesis, we also draw upon the theory of semi-Markov decision processes (SMDPs) [7, 29, 33, 44, 69]. Actions in SMDPs take variable amounts to time to complete. We model our robot soccer and CTF plays as temporally-extended actions, since the amount of time needed to score a point when executing a play from a given state is drawn from an arbitrary distribution. Section 5.1 presents our formal definition of TRSMDPs; our formalism is slightly different than that of other authors because we measure the transition functions empirically, building models of our domains' transition dynamics offline. Given a state/action pair (s, a), the world produces an *outcome* (s', dt, r). Our estimated transition function $\hat{T}(s, a)$ therefore returns a list of tuples of the form (p, s', dt, r), where p is the probability of transitioning to state s' after dt time steps, receiving reward r, when action a is executed in state s.

8.2 Decision Problems with Alternative Objective Functions

Bacchus et al. introduced *non-Markovian rewards* as a way to assign rewards to behaviors that extend over time [2]. They define non-Markovian reward decision processes (NMRDPs) as a generalization of MDPs in which the reward function R takes in *histories*, of the form $\langle s_1, s_2, \ldots s_n \rangle$. Since the explicit specification of such a reward function is impossible (there are an infinite number of possible histories), the authors use a temporal logic (PLTL) to specify non-Markovian rewards. Since the value of an action depends on history, policies in an NMRDP are a mapping from histories to actions. The authors present an algorithm for converting an NMRDP into a standard MDP by "expanding" the MDP: annotating each state with the additional history information needed to ascribe the rewards. There is a tradeoff between the effort spent translating the MDP—producing a *small* equivalent MDP without storage of irrelevant history—and the effort required to solve the result (since an MDP with many unneeded states will take longer to solve using standard MDP solution methods). Unfortunately, in the worst case, even the minimal expanded MDP may be exponentially larger than the base MDP. Some solution approaches specifically target factored MDP representations or anytime heuristic search algorithms [1, 71, 72]. Thresholded rewards are closely related to NMRDPs, as the thresholded-rewards objective function is also a form of non-Markovian reward. However, with thresholded rewards, we are not interested in accumulating rewards based on the past history of the system. Instead, we are interested in choosing actions based on the expected reward accumulated at some fixed point in the future. We use a similar MDP-expansion technique to solve TRMDPs; however, the size of our transformed MDP is only polynomially larger than the original MDP. In this thesis, we present a value iteration algorithm that exploits the structure of our transformed MDP to find the optimal policy in polynomial time.

Liu and Koenig discuss risk-sensitive utility functions for MDPs [30–32]. In this framework, agents act to optimize utility, which is a monotonically nondecreasing function of reward. These utility functions capture the risk attitudes of human decision makers. For example, concave utility functions characterize risk-aversity; linear utility functions correspond to the standard "maximize expected rewards" objective. Liu and Koenig provide exact algorithms for finding optimal policies in domains with exponential utility functions and with one-switch utility functions (which are combinations of a linear utility function and an exponential utility function.) They also provide a functional value iteration algorithm that finds approximately optimal policies for domains with piecewise linear utility functions. However, Liu and Koenig do not explicitly address finite-horizon domains nor provide tight (e.g. polynomial) runtime bounds on their solution algorithms.

Wagman and Conitzer apply a thresholded-rewards objective function to a one-shot strategic betting domain [78]. In their domain, each agent i chooses a lottery (probability distribution) over non-negative numbers with an expected value equal to its budget b_i. Each agent then receives reward according to its chosen probability distribution. The goal is to be the agent with the highest outcome; an agent is assigned true reward of 1 for a win and 0 otherwise. Wagman and Conitzer show that, if all n agents' budgets are equal, there is a unique symmetric Nash equilibrium which maximizes the probability of winning for all players. If all players play the equilibrium strategy, each player has a $1/n$ chance of winning. Wagman and Conitzer further generalize their results to domains in which agents have unequal budgets, costly budgets, and/or private budgets.

Broz et al. utilize semi-Markov POMDPs [21] to model the time-dependent behavior of humans and robots in human-robot interaction domains [9]. They utilize a similar approach to our own, adding time as a variable to the states of the POMDP. The resulting model can be quite large, so Broz et al. introduce a state aggregation approach which operates exclusively in the time dimension of the state space. States that are likely to lead to similar future rewards are combined together. (See e.g. [28] for further discussion of state aggregation techniques in MDPs.) Broz et al. provide experimental results, similar to ours in Chapter 4, that show how the value of the policy in their domain decreases depending on the level of time compression.

Sutton and Barto present a blackjack domain which is similar to a thresholded-rewards problem [68]. In the blackjack domain, each hand proceeds until the end of the game, at which time the agent receives a score of $-1, 0$, or $+1$ based on the current state. Each game is of a finite (though not fixed) length. However, there is no notion of time in this domain—the optimal policy is completely determined by the current state and is stationary with respect to the amount of time that has passed. There is also no notion of intermediate reward—all the reward is given at the end of a hand. In a thresholded-rewards problem, we are instead interested in achieving a given intermediate reward within a finite amount of time. We can view the blackjack domain as thresholded-rewards by setting an objective function like the following: "Play 1000 hands of blackjack such that the probability of winning at least 500 hands is maximized." With this objective function, we would expect that the blackjack-playing agent's strategy would change during the course of the 1000 hands, depending on the actual wins and losses experienced so far.

8.3 Multi-Robot Teamwork

There is a wide variety of related research in the general fields of multi-robot teamwork and coordination, and in the specific field of robot soccer in particular. Balch and Parker survey a wide variety of multi-robot research, providing an overview of theoretical results as well practical algorithms as implemented in a variety of real-world robot teams [3]. Dudek et al. present a taxonomy of existing multi-robot systems, categorizing systems based on whether the robots are centralized or decentralized, homogenous or heterogenous, the level of communication used by the robots, and so on [10]. Gerkey and Mataric present a taxonomy of multi-robot task allocation problems that is based on how many tasks a robot can execute concurrently, how many robots are needed to complete each task, and whether the tasks are assigned instantaneously [14, 15]. Our own role assignment algorithm is presented in Section A.2 of Appendix A.

Many researchers have used multi-robot teams to address tasks such as providing surveillance of a given area, monitoring the environment, or tracking multiple moving targets (e.g., [17, 20, 43, 60, 64–66, 73]). In Section 6.1 we present activity-recognition results from our team of soccer-playing robots. Unlike most of these other approaches, our team performs *incidental* activity recognition while engaged in a primary task other than monitoring the other agents in the environment. Our robots' observations of the environment and opponent robots happen by "accident" as the team plays soccer. Compared to other approaches in which the primary task is to investigate the environment, our RoboCup team does not explicitly track the movements of other robots in the area nor attempt to maximize the portion of the environment that is viewed.

8.4 Teamwork in Robot Soccer

Gerkey and Mataric discuss role assignment in robot soccer as of 2003, giving an overview of the strategies used by teams in each of the RoboCup leagues [16]. In the four-legged league, the predominant approach involved allocating three roles: an *attacker*, a *defender*, and some sort of *supporter*. According to Gerkey's survey, nearly all RoboCup teams assigned roles to robots in a greedy fashion. For example, prior to 2004, our own team (CMPack) assigned the roles in a fixed order using a well-defined objective function, namely first the *attacker* role to the robot that could reach the ball most quickly, then the *defender* role to whichever of the other two robots was closest to our own goal, and finally the *supporter* role to the remaining robot [74]. To prevent robots from interfering with one another, the attacker was the only robot allowed to actually approach the ball for a kick; the other robots would position themselves in useful supporting locations. If the ball came near to another robot, the team members would negotiate a role switch. After the role switch, the closest robot to the ball would become the new attacker, and the attack would continue.

Many approaches to teamwork in RoboCup utilize global potential fields for robot coordination. Veloso et al. present an algorithm called SPAR (Strategic Positioning using Attraction and Repulsion) which uses potential fields to repulse each robot from opponents and teammates, to attract each robot to the opponent goal and to the current position of the ball, and to position in areas where it is possible to successfully receive a pass. The SPAR algorithm was first used by CMU's entries to the RoboCup simulation and small-size leagues in 1998, and for several years thereafter. Prior to the introduction of plays, our CMPack AIBO team also used potential fields to position the defender and supporter robots [74]. Potential field approaches have also been utilized in the RoboCup middle-size league (e.g., [80]) and by other teams in the four-legged league (e.g., [27]).

Bowling et al. introduce a play-based approach to teamwork for the RoboCup small-size league [5, 8]. A play specifies a *plan* for the team; i.e., under some applicability conditions, a play provides a sequence of steps for the team to execute. A team can be equipped with multiple plays that achieve the same overall objective in different fashions; Bowling et al. show that play selection weights can be adapted online to learn which alternatives work well against a specific opponent [6]. Miller et al. have investigated a play-based approach for human-robot interaction [40]. The small-size league has centralized control, an overhead camera that provides an unobstructed view of the entire environment, and very fast and precise low-level robot actions. In this thesis, we introduce a play-based approach to teamwork for the RoboCup four-legged league. Since each robot in the four-legged league is fully autonomous and there is no centralized control, our teamwork algorithms are designed to operate within the constraints of a distributed team. Since the four-legged league suffers from significantly more sensor and actuator noise than the small-size league, and extensive communication is costly, we have designed our approach so that team members operate in a relatively loosely-coupled and autonomous manner. Unlike other approaches, we do not make use of global potential fields, but instead employ a region-based approach in which each robot is assigned a region of the field and positions itself within that region in a role-dependent manner.

After observing our play-based teamwork approach at the RoboCup 2005 world competition, Quinlan et al. [48–50] developed a similar system for the NUbots team in 2006. Quinlan et al. provide rules for autonomous strategy selection which are implemented as hand-coded if-statements in the Python programming language, such as:

```
if (secondHalf and (oppScore > ownScore)):
    scoreDiff = oppScore - ownScore
    if (timeLeft/60.0 < scoreDiff):
        newStrategy = AGGRESSIVE
```

Similar to our own region-based roles, the system presented by Quinlan et al. also assigns robots to specific areas of the field. They test three plays experimentally: *sweeper*, which is the play primarily used by NUbots in competition; *offensive*, in which all field players move near the opponent goal; and *hold*, in which two players adopt defensive roles. They find that the offensive play suffers from severe weaknesses, namely that the lack of defenders makes it very difficult to score. The sweeper and hold plays have roughly similar performance. In their experiments, Quinlan et al. use only goal count and shots on goal as their performance metrics, and do not provide statistics on the time-to-score distribution for each play combination or consider how play switching can be done in a manner that maximizes the probability of winning the game.

Dylla et al. propose a soccer strategy language that formalizes the strategies and tactics used by human soccer teams [11, 12]. Their language allows soccer strategies to be specified in an abstract way that does not depend strongly on the specific robot hardware used in the competition. Our play system can be viewed as a strategy language that is applicable to any distributed multi-robot system but that has been specifically tested in the RoboCup four-legged league.

8.5 Strategic Decisions in American Football

The game of American football presents many different types of strategic decisions which must be made by coaches on the field. Since football is a timed, zero-sum game, the optimal decision in many situations depends on score and time. Many researchers have approached these decisions in a theoretical fashion, finding optimal policies for specific decisions that football coaches face.

Sackrowitz addresses the points-after-touchdown decision: after a touchdown, should a team attempt to kick an extra point or go for a two-point conversion? [59] In practice, most coaches make their decision based only on the score difference. For example, if the coach's team is ahead by 12 points after the touchdown, the coach usually chooses to go for a two-point conversion. If the two-point conversion is successful, then the team will be ahead by 14 points (which is usually equal to two touchdowns). Sackrowitz introduces a dynamic-programming model of American football games which aims to maximize the probability of winning. According to this model, the optimal policy depends on time as well as score. For instance, early in the game, a team that has just scored a touchdown and is now 12 points ahead should kick the extra point rather than attempting a two-point conversion. Instead of using real time, Sackrowitz's model uses the number of possessions left in the game as the time horizon. This is because otherwise they would need to estimate a probability distribution for the length of a possession. Given our work on TRSMDPs, we could model the domain more accurately by actually representing the length of a possession explicitly in the model.

Krasker addresses three distinct strategic decisions in American football: whether to attempt a two-point conversion, whether to try an onside kick, and whether to use a hurry-up offense [26]. This work attempts to maximize the probability of winning, and the value function depends on the time remaining. However, the model assumes that the time taken for a drive follows a log-normal distribution; our work with SMDPs would let us accurately model the true drive-length distribution as measured from empirical data. For the hurry-up offense decision, they assume that the offense definitely finishes the drive before the end of the game, no matter how much time is left; however, the less time is left, the lower the chance of successfully scoring. With SMDPs, we could more accurately model the hurry-up offense; we could also have drives that are explicitly meant to eat more time off the clock than usual.

Romer [57] and Patek and Bertsekas [45] analyze additional strategic decisions in American football, including choosing the optimal play on fourth down and maximizing the expected score of a single drive. However, these researchers do not model score and time: they are concerned only with maximizing expected score. While interesting, the work presented in these papers is not immediately relevant to our goal of acting so as to maximize the probability of winning.

8.6 Summary

In this chapter, we discussed lines of related research, including the theory of Markov decision processes (MDPs) and semi-Markov decision processes (SMDPs), decision problems with alternative objective functions, multi-robot teamwork, and approaches to teamwork and strategy in robot soccer and American football.

Chapter 9

Conclusion

In this chapter, we conclude the thesis by reviewing the major scientific contributions of this thesis and discussing promising directions for future research.

9.1 Contributions

This thesis makes the following scientific contributions:

- We formally define *thresholded-rewards problems* as a means to analyze the tradeoffs between maximizing score and maximizing the true objective function (e.g., the probability of winning), in domains with limited time and some notion of score, progress, or intermediate reward. In a domain with thresholded rewards, *intermediate rewards* are received during execution. At the end of the time horizon, *true reward* is assigned by applying an arbitrary user-defined threshold function to the total intermediate reward accumulated during execution. The policy that maximizes the expected value of the zero-sum threshold function (Equation 3.1) is the policy which maximizes the probability of winning a timed, zero-sum game. Similarly, the zero-one threshold function (Equation 3.2) is used to find a policy which maximizes the probability of achieving reward of at least k before the time horizon elapses.

- We present an algorithm which finds optimal policies for thresholded-rewards MDPs. This algorithm takes an input a base MDP modeling the underlying dynamics of the domain, a reward threshold function, and the length of the time horizon. The algorithm creates an expanded MDP in which score and time are explicitly represented. Applying value iteration to the expanded MDP efficiently finds the solution, producing the optimal policy for the thresholded-rewards problem.

- We introduce three heuristic approximation techniques that find approximate solutions to TRMDPs. These heuristics trade off computation time with solution quality, achieving performance close to the optimal solution while using significantly less computation time.

- We present an exact algorithm for solving thresholded-rewards SMDPs. TRSMDPs accurately model domains in which actions are temporally extended or in which the amount of time taken to achieve a reward is drawn from an arbitrary distribution.

- We introduce *incidental behavior recognition* as an interesting problem arising in domains with limited time and in which the primary task is not just to classify the behavior of other agents operating in the same environment.

- We analyze how a team should change strategy in response to an opponent whose behavior is initially unknown but slowly reveals itself during execution.

- We introduce a sampling-based control algorithm that allows for effective action in domains with unknown rewards, which are hidden from the agent. We show that reasoning about time and score can lead to a significant benefit, even if we only obtain a small sample of reward values during execution time.

- We apply and evaluate our techniques in three different timed, finite-horizon domains: robot soccer, Capture the Flag, and reCAPTCHA. With our real robot soccer team, we perform proof-of-concept experiments that show that the choice of different plays leads to significantly different outcomes against an opponent. We also test incidental behavior recognition on the real robots, showing that the team can distinguish between opponent plays despite the significant sensor noise experienced by the robots. We use CTF extensively to find significant results supporting our TRSMDP approach, and also to analyze how a team should behave when facing an opponent whose strategy is initially unknown. Finally, we analyze 29 million human answers collected over six months to produce a model of the reCAPTCHA domain. The reCAPTCHA domain shows that our algorithms also apply to limited-time domains in which there is no adversary. We also validate our sampling-based control algorithm in the reCAPTCHA domain.

9.2 Future Directions

We enumerate a number of directions for future work closely related to the technical contributions of this thesis.

- **Full integration with the real robot soccer team.** In this thesis, we measured the time-to-score distributions for two different plays in our real robot soccer team; based on these time-to-score distributions, we computed optimal policies for the robot soccer domain. However, we have not performed any real-robot experiments which empirically verify that our team actually achieves significantly more wins than losses when playing the optimal policy. It would also be better to include more plays so that the team has a greater variety of potential actions. Furthermore, it would be better to collect more data and obtain more accurate time-to-score distributions. The main problem here is simply one of logistics: in order to incorporate n plays, we need to measure $O(n^2)$ time-to-score distributions; in order to find more accurate time-to-score distributions, we need to run more games in each condition; and in order to find that our team wins significantly more games, we need to run many games playing the optimal strategy against each possible play. Since these games all need to be refereed by a human, it is very time-consuming to run all these trials (in fact, competition games are refereed by four humans). Furthermore, our robots are not designed to handle playing for such extended periods of time—three robots were broken during the course of the experiments presented in this thesis.

- **Modeling the cost of switching plays.** In both robot soccer and CTF, we have observed that there is a cost to switching plays. Team members require time to reconfigure themselves for their new roles. During this period of transition, the team can be somewhat vulnerable, as multiple team members are out of their expected positions. How can we model the fact that play transitions are not instantaneous? We could consider measuring the time-to-score distributions induced during each transition from p_x to p_y and explicitly factoring these into our model. We designed our robot soccer team to limit the cost of play switching where possible, but we could consider further research into how a team (in general) can transition smoothly from one team-level behavior to another.

- **The cost of imprecise opponent modeling.** In this thesis, we assume that the opponent's possible play choices are the same as ours. In reality, some opponents can certainly be characterized as more aggressive or more defensive than others, but it is unlikely that their plays will be exactly the same as our own. Since there isn't enough time to learn the full opponent behavior online, our team will need to find the best match between the opponent's actual behavior and the closest opponent model we have available. How much do we lose due to the fact that our model of the opponent's behavior is imperfect?

- **Hierarchical reward threshold functions.** In some domains, we are presented with thresholds that are applied hierarchically or repeatedly. For example, in order to win a single-elimination tournament, each individual game must be won. Rather than maximizing the probability of winning a single game, we should aim to maximize the probability of winning the entire tournament. If individual games are not independent, or if there are actions that can be taken between games, it may be the case that the optimal strategy is to purposely decrease the probability of winning one game in order to increase the chances of winning future games. For example, perhaps playing as hard as possible in an "easy" game increases the chance of injury; or perhaps we are hesitant to reveal some strategy or capability early in the competition, thereby giving future opponents the chance to adapt. Can we extend the algorithms presented in this thesis to find optimal policies for domains with hierarchical thresholding?

- **Incidental behavior recognition.** In this thesis, we show that it is possible to recognize the behavior of the opponent team incidentally, even though our team takes no explicit actions in order to gather more information about the behavior of the opponent team. This general problem of recognizing the behavior of an opponent online (or of learning other things about the environment online), while being fully engaged in a primary task of greater importance, leads to many open questions that touch on the fields of multi-agent systems, model-based learning, and POMDPs. In a domain with limited time, how does the accuracy of behavior recognition affect the probability of successfully completing the primary task? Should our team trade off some short-term performance in order to improve our model of the world and achieve better long-term performance? In a multi-agent or multi-robot team, a team could consider allocating some agents to perform the primary task and allocating other agents to assess the performance of the others or to learn more about the environment. What is the best way to do this allocation?

- **Partially observable domains.** In this thesis, we generally assume that the useful state features for team-level strategy selection are significantly abstracted at a high level, and that the state of the world is effectively fully observable. Clearly, these assumptions are not valid in some domains—particularly when trying to recognize the behavior of an opponent online. In such cases, the ideal approach would be to model the domain as a POMDP with a thresholded-rewards objective function. We could then use techniques similar to those introduced in this thesis, creating an expanded POMDP annotated with score and time, in order to calculate optimal policies for partially observable domains.

- **Convergence results.** In this thesis, we generally assume that the behavior of the opponent is static—though in Chapter 6 we do consider a simple case of an opponent that changes strategy during the course of a game. What is the effect if the opponent changes strategy fully, based on score and time, as our team does? If the opponent knows that our team will become more aggressive because our team is losing, perhaps the opponent will become more defensive in response. If so, our team might increase the probability of winning by becoming aggressive even sooner; then the opponent may become defensive even sooner, and so on. If both sides apply this reasoning repeatedly, there should be convergence to some optimal policy in which each side has the same probability of winning. How can this policy be derived, and what will it look like in different domains? There is a chance that the optimal policy could diverge with respect to time, degenerating to something like "when winning, choose the most defensive play; when losing, choose the most aggressive play; when tied, choose some balanced play."

9.3 Concluding Remarks

In this thesis, we have contributed several algorithms which aim to address the challenges of acting effectively in complex stochastic domains with limited time and some notion of score. We have showed that the thresholded-rewards objective function is applicable to a variety of decision problems, including zero-sum games and domains in which we aim to achieve a given amount of reward. The solution to a thresholded-rewards decision problem provides an optimal policy which enables an agent or team of agents to maximize the probability of achieving their goals before the time deadline expires. The work presented in this thesis has broad applications to domains possessing the key features of control under uncertainty, limited time, and some notion of score.

Appendix A

Communication and Play-Based Role Assignment in the RoboCup Four-Legged League

In this appendix, we present additional details on the communication and coordination strategies employed by our CMPack and CMDash entries to the RoboCup Four-Legged League from 2004–2008. Although these algorithms and results are not directly related to the main contributions of the thesis, we present them here in order to provide additional groundwork and context for understanding our approach to robot soccer teamwork.

Section A.1 presents an overview of the various communication messages broadcast by each member of our distributed team. These messages are used to build a world model that is shared among all team members. Section A.2 presents full details of our distributed, play-based role assignment approach for the RoboCup Four-Legged League. Section A.3 presents experimental results showing that our role allocation algorithm assigns roles to robots in a manner that is resistant to role oscillation.

A.1 Communication Strategies

Many multi-robot teams make use of communication for world state sharing. Due to the AIBOs' limited perception range and the extensive object occlusion in the RoboCup environment, teams can benefit greatly by building a shared world model. A common approach, used by our team in the past [58], is to have each robot periodically broadcast a packet containing all the shared information, such as the robot's current position, its best estimate of the ball position, and the positions of detected opponent robots. However, some domain information (such as the position of the ball) is inherently more important to the success of the team than other types of information. We have therefore developed a factored communication strategy. In this strategy, there are several different types of message, containing different pieces of information. We can then independently choose the transmission rate for each type of message. This communication strategy allows the robots to respond more quickly to important events (such as a change in the ball's position) without the need to transmit a large message over the communication network. We present here a brief overview of the world-modeling information our robots shared in the RoboCup 2005 competition.

A.1.1 Ball Messages

These messages are sent by all robots to indicate important information about the status of the ball. Each message contains the following information:

- Ball state. This feature was added to our world model for RoboCup 2005. This can take on one of the following values:
  - Lost: No reliable estimate of the ball's location is available.
  - Visible: The ball is currently seen.
  - Possession: The ball is believed to be in the possession of the robot (i.e., the robot has grabbed the ball and is lining up for a kick.)
  - NotInFOV: The ball is not currently seen, but is not expected to be seen because it is outside the robot's field of view. This happens (e.g.) when a robot takes its view off the ball to look at a localization marker.
  - InFOVButMissing: The ball is not currently seen, even though the robot believes it is looking at the ball's location.
  - InFOVButOccluded: The ball is not currently seen, but the robot believes that an object (such as another robot) is occluding the ball.

  Our team sends these symbolic ball states instead of numerical confidence values. These symbolic values allow the team to more accurately characterize the true state of the ball.

- Whether the robot transmitting the message believes that it is lost. This is determined by thresholding the robot's localization uncertainty.

- The global position of the ball. Global ball position estimates are not used from any robot that claims to be lost, since a lost robot is very likely to project its local ball estimate to an incorrect global position.

- The position of the ball relative to the robot. If the ball is very close to the reach of a robot, and that robot intends to kick the ball, the robot's teammates should avoid interfering with the kick, even if the robot believes that it is lost. The transmission of relative ball locations allows robots to back off in this situation, without the need to rely on visually seeing the teammate near the ball.

Since the location of the ball is of utmost importance to the proper functioning of the team, the ball messages are sent frequently. A robot will send a new ball message every 1/8 second if it has a good ball hypothesis and is not lost. If the robot becomes lost or does not have a valid ball hypothesis, it waits a while longer to see if the situation improves. This is done because a valid global ball location provides more valuable information to teammates. After 1/4 second has passed, however, the robot sends a ball message regardless of the circumstances.

A.1.2 Status Messages & Intentions

Another type of message is the status message. Status messages are sent by each robot at periodic intervals (typically 4 Hz). They include the robot's current position and angle (as reported by localization) as well as the current "intention" of the robot. Intention is a very important concept that we have added to our teamwork strategy. When a robot is very close to the ball, its teammates should stay out of the way, to ensure that they do not interfere with the attacker's actions. However, there are specific times when nearby robots might not be intending to go for the ball. In these cases, the teammates should not back away just because another robot is near. The intention of the robot is determined by the robot's top-level behavior, and can take on any of the following values:

- ATTACK: the robot intends to approach the ball and manipulate it.

- WAIT: the robot does not intend to approach the ball. This happens when a robot is returning to position or is searching for the ball.

- YIELD: the robot would intend to approach the ball, except that it is yielding to a teammate instead.

A.1.3 Periodic Messages

Periodic messages are provided as a form of robustness to failure. The information contained in periodic messages allow the robots to determine when network failures have occurred, when a teammate has crashed, or other anomalous events have occurred. The team can then take appropriate actions to ensure that team play degrades gracefully in the presence of failure. The periodic message is typically sent at a rate of 1 Hz.

A.2 Distributed Play-Based Role Assignment

It is our experience that it is rather challenging to generate or learn a team control policy in complex, highly dynamic (in particular adversarial), multi-robot domains. Therefore, instead of approaching teamwork in terms of a mapping between state and joint actions [47, 70], we follow a *play-based* approach, as introduced by Bowling et al. [5,6,8]. A play-based approach allows us to handle the domain challenges introduced in section 2. A play specifies a *plan* for the team; i.e., under some applicability conditions, a play provides a sequence of steps for the team to execute. Multiple plays can capture different teamwork strategies, as explicit responses to different types of opponents. Bowling showed that play selection weights could be adapted to match an opponent. Plays also allow the team to reason about the zero-sum, finite-horizon aspects of a game-playing domain: the team can change plays as a function of the score and time left in the game. Our play-based teamwork approach ensures that robots do not suffer from hesitation nor oscillation, and that team performance is not significantly degraded by possible periods of high network latency. We believe that ours is the first distributed play-based teamwork approach within the context of the RoboCup four-legged league.

A.2.1 Plays

A *play* is a team plan that provides a set of roles, which are assigned to the robots upon initiation of the play. Bowling [6] introduced a play-based method for team coordination in the RoboCup small-size league. However, the small-size league has centralized control of the robots. One of the significant contributions of our work is the development of a play system that works in a distributed team. The play language described by Bowling assumes that the number of robots is fixed, and therefore always provides exactly four different roles for the robots. In another extension to Bowling's work, our plays also specify which roles are to be used if the team loses some number of robots due to penalties or crashes. This extension to the role-assignment aspects of Bowling's play language allows the team to robustly adapt to the loss or penalization of team members without the need for additional communication. This is a particularly important extension for domains in which limited or high-latency communication is the norm.

Our play language itself is also strongly inspired by the work of Bowling. Our language allows us to define *applicability conditions*, which denote when a play is suitable for execution; what *roles* should be assigned when we have a specific number of active robots on the team; and a *weight*, which is used to decide which play to run when multiple plays are applicable.

Applicability. An applicability condition denotes when a play is suitable for execution. Each applicability condition is a conjunction of binary predicates. A play may specify multiple applicability conditions; in this case, the play is executable if any of the separate applicability conditions are satisfied.

Roles. Each play specifies which roles should be assigned to a team with a variable number of robots by defining different `ROLES` directives. A directive applies when a team has k active robots, and specifies the corresponding k roles to be assigned. If a robot team has n members, each play has a maximum of n `ROLES` directives. Since our AIBO teams are composed of four robots, our plays have four `ROLES` directives.

Weight. Weight is used to decide which play to run when multiple plays are applicable. In our initial approach, the play selector always chooses the applicable play with greatest weight. We could also consider choosing plays probabilistically based on the weight values or updating the weights at execution time to automatically improve team performance. *Playbook adaptation* of this sort was introduced by Bowling et al. for the small-size league [6].

Unlike the work of Bowling, we do not have `DONE` or `TIMEOUT` keywords that specify when a play is complete. Rather, the play selector runs continuously, and each play is considered to be complete as soon as a different play is chosen. This may happen because the current play is no longer applicable or because another play with greater weight has recently become applicable. Each predicate used in an applicability condition is designed with some hysteresis, such that it is not possible for the predicate to rapidly oscillate between true and false. The predicates used in our approach depend on features of the environment—such as the time left in game, the number of goals scored by each team, and the number of robots available to each team—that by their nature cannot rapidly oscillate. This ensures that the play choice also cannot rapidly oscillate.

```
PLAY Guard
APPLICABLE winning fewerPlayers
APPLICABLE secondHalf winningBy2OrMoreGoals
ROLES 1 Goalkeeper
ROLES 2 Goalkeeper Defender
ROLES 3 Goalkeeper Defender Independent
ROLES 4 Goalkeeper Defender Midfielder Independent
WEIGHT 3
```

Figure A.1: An example play with multiple applicability conditions.

```
Default: Goalkeeper Defender Striker Independent
Defensive: Goalkeeper Defender Midfielder Independent
Guard: Goalkeeper Defender MidfieldDefender Independent
Flankers: Goalkeeper Defender LeftFlanker RightFlanker
Aggressive: Goalkeeper LeftFlanker RightFlanker Independent
PullGoalie: Midfielder LeftFlanker RightFlanker Independent
Kickoff: Goalkeeper Defender Charger KickoffDodger
```

Figure A.2: Summary of the seven plays used by our team in RoboCup 2005.

Figure A.1 shows an example of a defensive play. Its applicability conditions specify that this play is applicable 1) when our team is winning and has fewer active players than the opponents or 2) when the game is in the second half and our team is winning by at least two points. If we have only one active robot on our team, we will assign it the Goalkeeper role; if we have two robots, one is assigned the Goalkeeper role and the other is assigned the Defender role; and so on. We have developed a total of sixteen plays, but not all were used in the RoboCup 2005 competition. Figure A.2 shows a summary of the seven plays that were used in the competition. (Only the roles used for a 4-robot team are shown.)

A.2.2 Play Selector

The *play selector* runs on one robot that is arbitrarily chosen to be the leader. The play selector chooses which play the team should be running. The leader periodically broadcasts the current play (and role assignments) to its teammates. Distributed play-based coordination is achieved through a predefined agreement among the team members to resort to a *default play* if a robot doesn't hear a play broadcast within a *communication time limit*. A

- **Game result predicates:**
  `winning`, `losing`, `winningByTwoOrMore`, `losingByTwoOrMore`.
- **Game state predicates:**
  `ourKickoff`, `opponentKickoff`, `fewerPlayers`, `morePlayers`.
- **Temporal predicates:**
  `firstHalf`, `secondHalf`, `<2MinutesRemaining`, `<1MinuteRemaining`, ...

Figure A.3: Predicates used in the applicability conditions of plays.

```
SELECT_PLAY(S: world state, P: playbook, D: default play):
  BEST_PLAY <- D
  BEST_WEIGHT <- WEIGHT(D)
  for each PLAY in P:
    if WEIGHT(PLAY) > BEST_WEIGHT:
      for each CONDITIONS in APPLICABLE(PLAY):
        if all CONDITIONS are satisfied in STATE:
          BEST_PLAY <- PLAY
          BEST_WEIGHT <- WEIGHT(PLAY)
  return BEST_PLAY
```

Figure A.4: Algorithm used by the play selector.

failure of the leader or a network problem may trigger this default coordination plan. A more sophisticated approach could incorporate an algorithm for *leader selection* in the event of failure. However, we did not pursue such an approach for the work presented in this thesis. The algorithm used by the play selector is presented in Figure A.4.

A.2.3 Role Allocator

The selection of a play determines which roles need to be allocated to the robots. However, it does not specify which robots should be assigned to each role. Therefore, a role allocation algorithm is still needed to assign the roles. This algorithm also runs on the leader robot, which broadcasts the assignment along with the selected play. Our role allocator has two features that differentiate it from those used by many other RoboCup teams [16]. First, it only runs when a play is initially selected, as opposed to continuously. Second, it allocates roles in a *role-preserving* manner – minimizing role switching. Formally, if a new play P_t is selected at time t, and P_t specifies n roles $\{R_{1..n}\}$ for the n robots $r_{1..n}$, and r_i was already assigned to R_j in P_{t-1}, r_i is guaranteed to still be assigned to R_j in P_t. (Any remaining roles can be allocated in a greedy fashion.) If two plays share some roles, this strategy guarantees that some of the robots can assume their new roles without any transitional cost. These features provide additional resistance to oscillation in cases in which two plays share common roles.

The play selector and role allocator combine to form our team's overall coordination strategy. The play selector PlaySel is a function that takes in the game state s and outputs a set R of roles. The role allocator RoleAlloc is another function; it takes in the state s and the set R of roles specified by the play selector, and assigns each role in R to a robot on the team. Formally, we have:

$$\Pi : s \rightarrow \textsc{RoleAlloc}(s, \textsc{PlaySel}(s)), \tag{A.1}$$

which evaluates to a vector of specific roles for each robot:

$$\Pi : s \rightarrow \langle r_1, r_2, \ldots, r_n \rangle. \tag{A.2}$$

A.2.4 Roles

The *role* assigned to each robot determines what behaviors the robot actually runs. Our approach, initially used in RoboCup 2005, is unique in that it is *region-based*: each robot is assigned to a region of the field. A robot is primarily responsible for going after the ball whenever the ball is in that robot's region. Roles are designed simply by configuring a generic "Player" behavior with appropriate settings for that role. The configurable items include:

- Region: an area of the field that the robot is responsible for covering.
- Ball in Region Policy: the behavior the robot should adopt when it knows that the ball is in its region. Typically, this will involve approaching the ball and trying to clear it downfield or to take a shot on goal.
- Ball out of Region Policy: the behavior the robot should adopt when it knows that the ball is not in its region. Some roles specify that a robot is simply to return to a home position, while other roles may have the robot move to block the path between the ball and the goal, or to position for a pass.
- Ball Lost Policy: the behavior the robot should adopt when it believes the ball is lost. This is typically some sort of searching behavior.

As an example of how these policies combine together to combine a complete player, we consider the behavior of the Defender role in detail. The defender's region includes the area outside the goal box but still near the goal. The defender will use ball estimates from any of its teammates, but puts preference on ball positions reported by the goalkeeper. When the ball is in its region, the defender approaches it in an attempt to clear. When the ball is not in its region, the defender takes a position between the ball and the goal, such that it is ready to interfere with any threats posed by the opponents. When the ball is lost, the defender returns to a home position in front of the goal and spins in place.

Unlike our previous approaches, robots no longer need to negotiate with one another in order to gain the *attacker* role that allows them to approach the ball. In this way, the performance of the team does not degrade significantly under high network latency. We have developed algorithms that prevent the robots from interfering with one another even when they are playing in overlapping regions. To provide robustness against communication failure, these algorithms are designed to operate without the need for communication, using local information such as a robot's vision of its own teammates. If communication is available, our robots use additional features (such as reported teammate positions) that provide added confidence that our robots will not interfere with one another.

A.3 Experimental Results

We have previously presented empirical results that support the feasibility and effectiveness of multiple plays in the RoboCup four-legged league [37]. In Section A.2.3, we contribute a role allocation algorithm, claiming that this algorithm addresses hesitation due to role oscillation by preserving a robot's role when possible. We show experimental evidence that supports this particular claim.

In each experimental trial, three robots work together in a robot soccer task, namely *ball advancement* – moving the ball towards the opposing goal as quickly as possible. Figure A.5 shows the initial position of the robots, from which the team advances the ball down the field towards the goal. A trial is considered complete when either a goal is scored, the ball advances past the opponents' back line, or the ball hits one of the goal posts. The time of each completed trial is measured.

We test the robots' teamwork in three different team play configurations: (i) a single *Defender-Striker-Independent* play; (ii) a single *Defender-Midfielder-Independent* play; and (iii) switching every five seconds between the two plays in (i) and (ii). Since these two plays share two roles (defender and independent), we expect that, even with frequent play

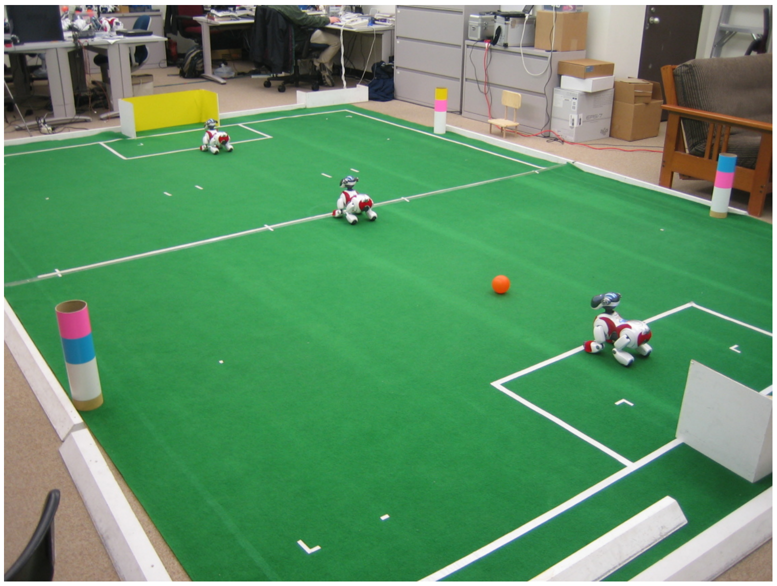


Figure A.5: Initial position for each experimental trial. The three robots are placed in three positions on the field, with the ball in the defense area. The experiment proceeds until the robots advance the ball past the end line of the opposite half of the field.

switching, our role assignment algorithm will not adversely affect the performance of the team.

Each configuration was tested for 40 completed trials, for a total of 120 experiments. Figure A.6 summarizes the results. The fastest and slowest times achieved in any trial were 17.18 and 70.15 seconds, respectively. The *Defender-Striker-Independent* play performs best at this task, completing each trial in a mean time of 31.06 seconds. The *Defender-Midfielder-Independent* play performs more slowly, completing each trial in a mean time of 35.05 seconds. The difference between these times is significant (determined by Student's two-tailed t-test, with $p = 0.048$). When the robots oscillate between these two plays, their performance remains good, with the mean time of the switching case (33.29 seconds) between the mean times of the other two cases. Since the play-switching case still performs better than the worse of the two plays, we note that there is no significant detrimental effect on performance.

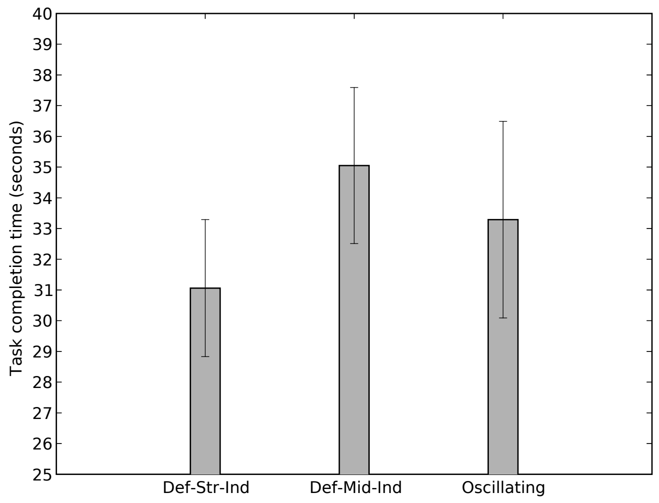


Figure A.6: Experimental results for the *Defender-Striker-Independent* play, *Defender-Midfielder-Independent* play, and switching between the two plays. The figure shows the means and 90% confidence intervals for each case.

A.4 Conclusion

In this chapter, we have presented the communication and world modeling strategies which were utilized by our robot soccer team from 2005–2008. These improvements place high priority on the communication of task-relevant data and enure that the robots communicate some useful information even when lost. We have also presented the details of a distributed play-based role assignment algorithm, which has been implemented on a distributed team of robots for the RoboCup four-legged league. This algorithm aims to solve several important general distributed multi-robot challenges, including the presence of adversaries, task-based temporal constraints, and robustness to network failure. We have presented experimental results that show that our role-preserving assignment algorithm allows a team to perform well even when plays are rapidly changed.

The presented role-assignment algorithm and plays have been tested in the RoboCup 2005 competition. Our team came in fourth place in a challenging competition of twenty-four teams. Our team typically rotated through three well-balanced plays in the first minutes of each game, which allowed us to see the performance of each play against the specific opponent. As a form of adjustable autonomy, we could manually change the team's strategy at halftime or during a timeout.

Our role assignment system is unique in that it allows role assignments to happen to all robots, including the goalkeeper. If there is not much time left in the game and our team is losing, we have plays that will “pull” the goalkeeper out of the goal box, which provides us with another field player that could score a goal. In fact, in the 3rd-4th place game of the RoboCup 2005 competition, our goalkeeper robot nearly scored a goal in the final seconds of the game.

Appendix B

Capture the Flag Experiment Data

In this appendix, we present the full results of our the Capture the Flag experiments. Section B.1 presents the results of our initial test of eleven CTF plays. Section B.2 presents the time-to-score distributions for each of 36 play combinations. Section B.3 presents the optimal policies against each of the six CTF opponents. Discussion and analysis of these results is presented throughout Chapter 5.

B.1 11-Play Preliminary Experiment Data

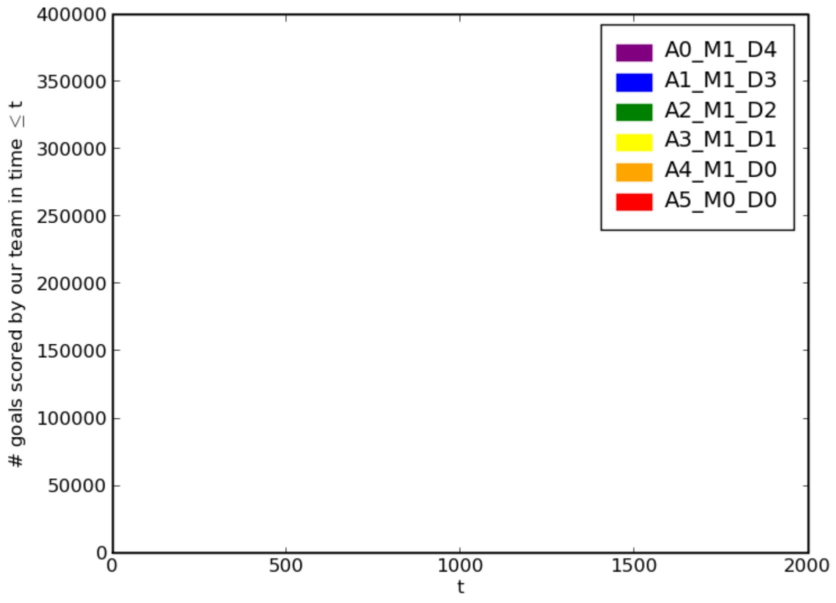
blue_config	red_config	blue_wins	red_wins	ties	blue_score	red_score
A0_M0_D5	A0_M0_D5	0	0	500	0	0
A0_M0_D5	A0_M1_D4	0	0	500	0	0
A0_M0_D5	A1_M0_D4	0	90	410	0	104
A0_M0_D5	A1_M1_D3	0	59	441	0	65
A0_M0_D5	A2_M0_D3	0	135	365	0	171
A0_M0_D5	A2_M1_D2	0	334	166	0	557
A0_M0_D5	A3_M0_D2	0	392	108	0	814
A0_M0_D5	A3_M1_D1	0	387	113	0	774
A0_M0_D5	A4_M0_D1	0	414	86	0	996
A0_M0_D5	A4_M1_D0	0	417	83	0	982
A0_M0_D5	A5_M0_D0	0	444	56	0	1208
A0_M1_D4	A0_M0_D5	0	0	500	0	0
A0_M1_D4	A0_M1_D4	0	0	500	0	0

A0_M1_D4	A1_M0_D4	0	37	463	0	37
A0_M1_D4	A1_M1_D3	0	32	468	0	32
A0_M1_D4	A2_M0_D3	0	110	390	0	126
A0_M1_D4	A2_M1_D2	0	227	273	0	306
A0_M1_D4	A3_M0_D2	0	326	174	0	557
A0_M1_D4	A3_M1_D1	0	332	168	0	559
A0_M1_D4	A4_M0_D1	0	381	119	0	767
A0_M1_D4	A4_M1_D0	0	384	116	0	771
A0_M1_D4	A5_M0_D0	0	415	85	0	986
A1_M0_D4	A0_M0_D5	77	0	423	100	0
A1_M0_D4	A0_M1_D4	42	0	458	46	0
A1_M0_D4	A1_M0_D4	150	145	205	455	485
A1_M0_D4	A1_M1_D3	18	247	235	51	449
A1_M0_D4	A2_M0_D3	86	295	119	559	1114
A1_M0_D4	A2_M1_D2	12	437	51	209	1633
A1_M0_D4	A3_M0_D2	64	382	54	1041	2179
A1_M0_D4	A3_M1_D1	28	429	43	587	2201
A1_M0_D4	A4_M0_D1	292	152	56	2886	2307
A1_M0_D4	A4_M1_D0	29	433	38	854	2565
A1_M0_D4	A5_M0_D0	464	21	15	5029	2188
A1_M1_D3	A0_M0_D5	65	0	435	76	0
A1_M1_D3	A0_M1_D4	40	0	460	43	0
A1_M1_D3	A1_M0_D4	255	18	227	497	64
A1_M1_D3	A1_M1_D3	35	54	411	52	72
A1_M1_D3	A2_M0_D3	251	53	196	566	168
A1_M1_D3	A2_M1_D2	127	139	234	251	279
A1_M1_D3	A3_M0_D2	325	65	110	1090	376
A1_M1_D3	A3_M1_D1	260	89	151	764	417
A1_M1_D3	A4_M0_D1	486	2	12	3315	397
A1_M1_D3	A4_M1_D0	269	100	131	1007	570
A1_M1_D3	A5_M0_D0	497	0	3	5991	333
A2_M0_D3	A0_M0_D5	148	0	352	196	0
A2_M0_D3	A0_M1_D4	104	0	396	122	0
A2_M0_D3	A1_M0_D4	278	85	137	1083	589
A2_M0_D3	A1_M1_D3	52	249	199	147	538
A2_M0_D3	A2_M0_D3	198	190	112	1151	1133
A2_M0_D3	A2_M1_D2	72	337	91	580	1400

A2_M0_D3	A3_M0_D2	235	202	63	1961	1889
A2_M0_D3	A3_M1_D1	111	313	76	1273	1964
A2_M0_D3	A4_M0_D1	457	21	22	4540	1977
A2_M0_D3	A4_M1_D0	116	318	66	1667	2490
A2_M0_D3	A5_M0_D0	495	1	4	6676	1725
A2_M1_D2	A0_M0_D5	331	0	169	546	0
A2_M1_D2	A0_M1_D4	225	0	275	296	0
A2_M1_D2	A1_M0_D4	447	11	42	1622	174
A2_M1_D2	A1_M1_D3	119	143	238	238	270
A2_M1_D2	A2_M0_D3	354	69	77	1448	583
A2_M1_D2	A2_M1_D2	203	177	120	697	672
A2_M1_D2	A3_M0_D2	396	54	50	2237	846
A2_M1_D2	A3_M1_D1	333	100	67	1589	870
A2_M1_D2	A4_M0_D1	497	0	3	5360	514
A2_M1_D2	A4_M1_D0	360	74	66	2121	1022
A2_M1_D2	A5_M0_D0	499	0	1	7838	278
A3_M0_D2	A0_M0_D5	364	0	136	681	0
A3_M0_D2	A0_M1_D4	332	0	168	533	0
A3_M0_D2	A1_M0_D4	359	70	71	2014	1028
A3_M0_D2	A1_M1_D3	57	323	120	391	1066
A3_M0_D2	A2_M0_D3	200	223	77	1883	1973
A3_M0_D2	A2_M1_D2	57	402	41	883	2213
A3_M0_D2	A3_M0_D2	226	218	56	2727	2669
A3_M0_D2	A3_M1_D1	66	389	45	1593	2994
A3_M0_D2	A4_M0_D1	458	24	18	5347	2393
A3_M0_D2	A4_M1_D0	65	394	41	1933	3547
A3_M0_D2	A5_M0_D0	497	1	2	7228	2011
A3_M1_D1	A0_M0_D5	378	0	122	771	0
A3_M1_D1	A0_M1_D4	328	0	172	515	0
A3_M1_D1	A1_M0_D4	430	31	39	2040	535
A3_M1_D1	A1_M1_D3	83	248	169	364	714
A3_M1_D1	A2_M0_D3	296	134	70	1892	1339
A3_M1_D1	A2_M1_D2	103	320	77	884	1524
A3_M1_D1	A3_M0_D2	385	72	43	2939	1614
A3_M1_D1	A3_M1_D1	181	224	95	1700	1792
A3_M1_D1	A4_M0_D1	499	0	1	6288	855
A3_M1_D1	A4_M1_D0	281	147	72	2447	1947

A3_M1_D1	A5_M0_D0	499	0	1	8442	503
A4_M0_D1	A0_M0_D5	422	0	78	1048	0
A4_M0_D1	A0_M1_D4	368	0	132	713	0
A4_M0_D1	A1_M0_D4	138	288	74	2207	2832
A4_M0_D1	A1_M1_D3	2	489	9	359	3320
A4_M0_D1	A2_M0_D3	31	454	15	1997	4530
A4_M0_D1	A2_M1_D2	0	498	2	485	5380
A4_M0_D1	A3_M0_D2	17	475	8	2397	5325
A4_M0_D1	A3_M1_D1	0	500	0	863	6340
A4_M0_D1	A4_M0_D1	241	211	48	4586	4499
A4_M0_D1	A4_M1_D0	0	500	0	1100	6902
A4_M0_D1	A5_M0_D0	450	31	19	6502	3442
A4_M1_D0	A0_M0_D5	406	0	94	963	0
A4_M1_D0	A0_M1_D4	398	0	102	745	0
A4_M1_D0	A1_M0_D4	435	27	38	2586	846
A4_M1_D0	A1_M1_D3	77	286	137	484	1089
A4_M1_D0	A2_M0_D3	312	132	56	2419	1757
A4_M1_D0	A2_M1_D2	63	380	57	1048	2165
A4_M1_D0	A3_M0_D2	393	68	39	3615	1995
A4_M1_D0	A3_M1_D1	153	286	61	1872	2433
A4_M1_D0	A4_M0_D1	500	0	0	6955	1055
A4_M1_D0	A4_M1_D0	202	236	62	2571	2594
A4_M1_D0	A5_M0_D0	500	0	0	8730	743
A5_M0_D0	A0_M0_D5	434	0	66	1227	0
A5_M0_D0	A0_M1_D4	407	0	93	896	0
A5_M0_D0	A1_M0_D4	12	470	18	2231	5065
A5_M0_D0	A1_M1_D3	1	496	3	312	5814
A5_M0_D0	A2_M0_D3	1	499	0	1751	6755
A5_M0_D0	A2_M1_D2	0	500	0	262	7892
A5_M0_D0	A3_M0_D2	1	497	2	1944	7325
A5_M0_D0	A3_M1_D1	0	500	0	444	8538
A5_M0_D0	A4_M0_D1	23	466	11	3371	6542
A5_M0_D0	A4_M1_D0	0	500	0	727	8763
A5_M0_D0	A5_M0_D0	222	250	28	5261	5352

B.2 Time-To-Score Distributions



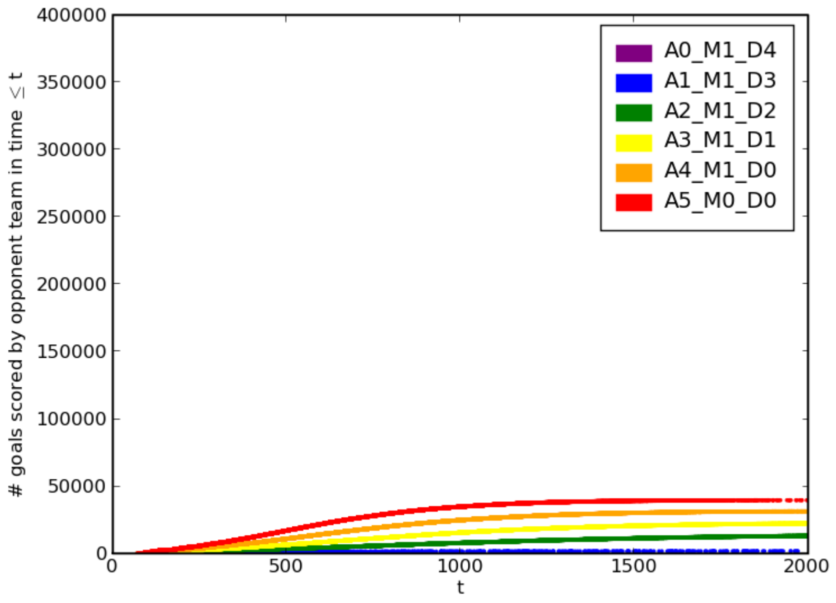
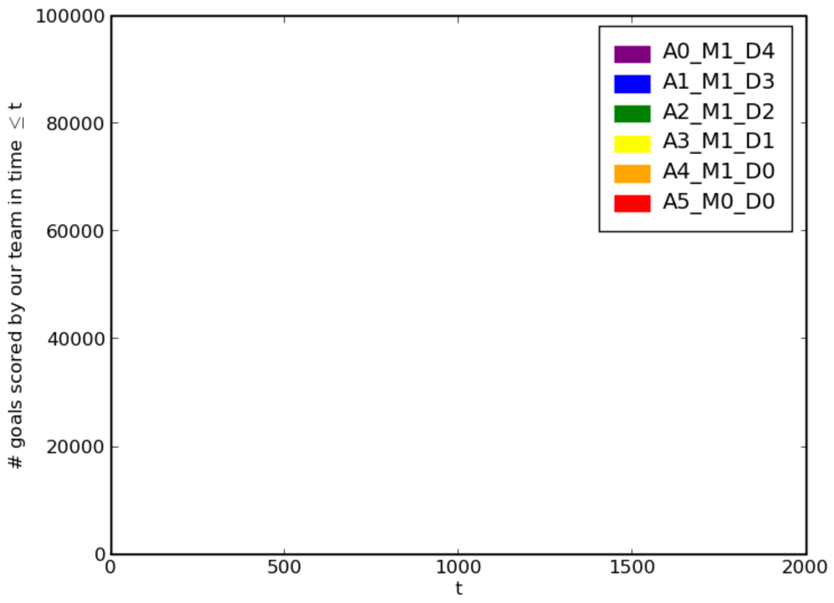


Figure B.1: Time-to-score distributions for the blue team (top) and for the red team (bottom) when the blue team plays `A0_M1_D4`, for each possible red play.



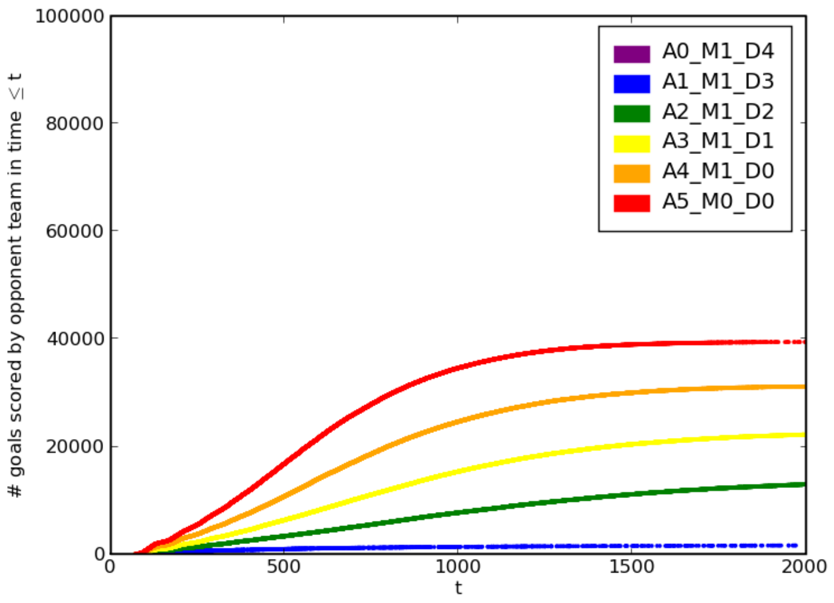
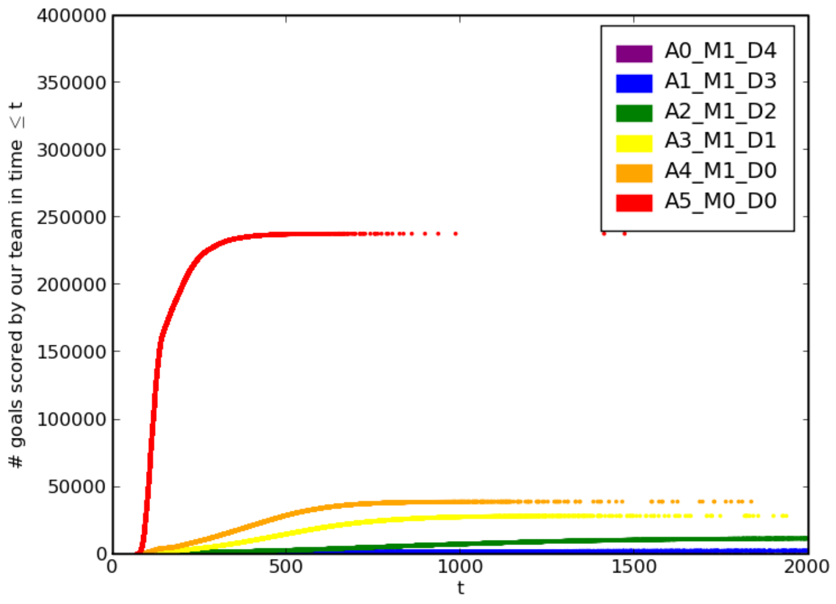


Figure B.2: Time-to-score distributions for the blue team (top) and for the red team (bottom) when the blue team plays `A0_M1_D4`, for each possible red play. This figure presents the same data as Figure B.1, but the y-axis is zoomed in.



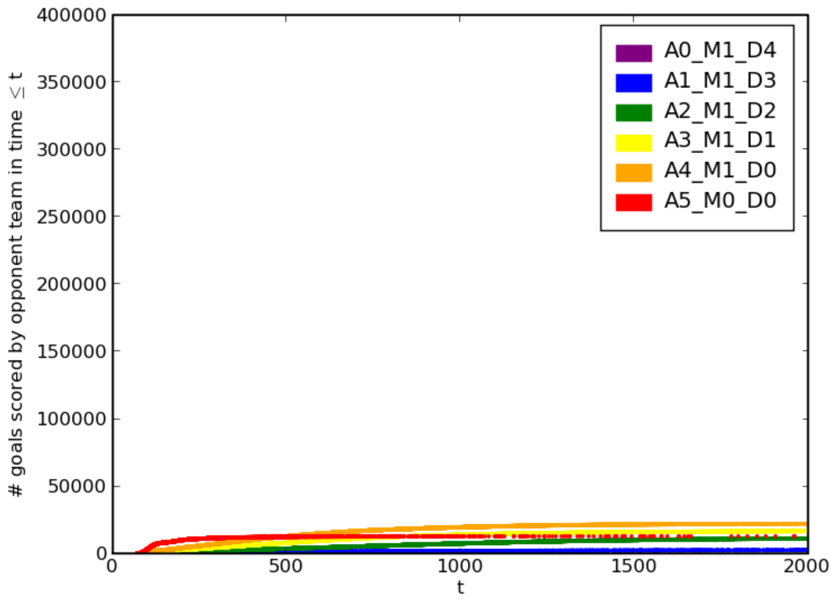
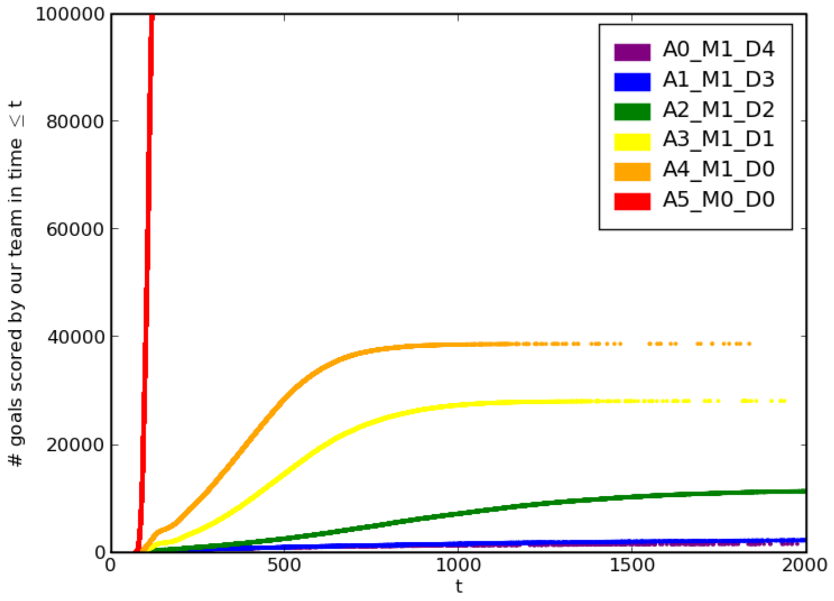


Figure B.3: Time-to-score distributions for the blue team (top) and for the red team (bottom) when the blue team plays `A1_M1_D3`, for each possible red play.



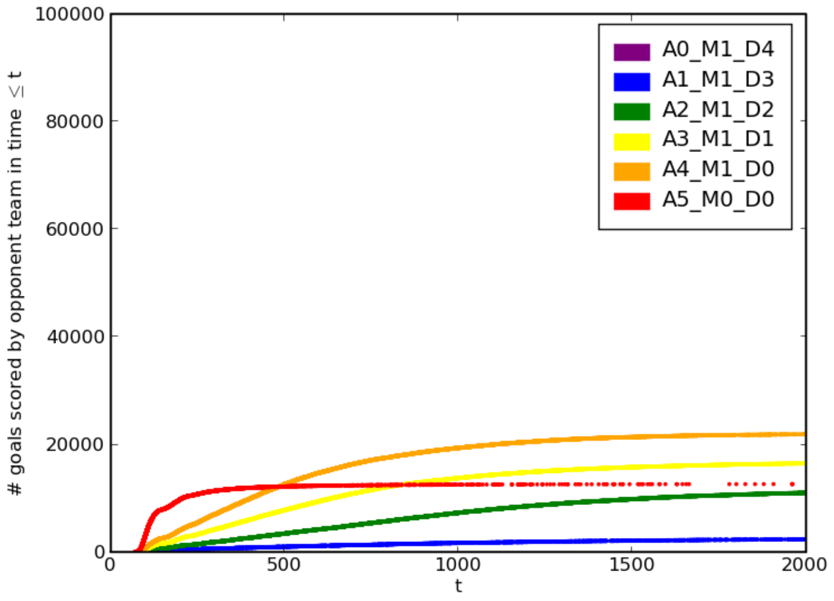
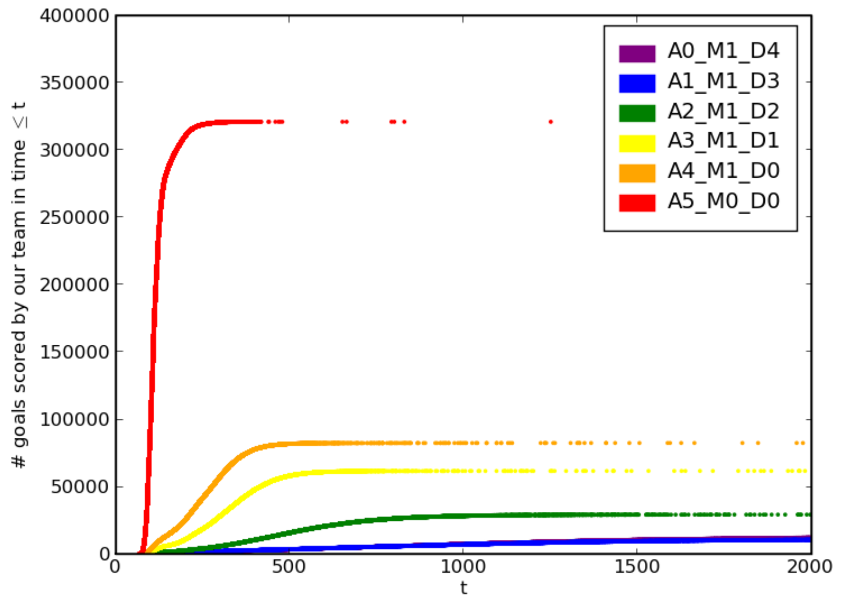


Figure B.4: Time-to-score distributions for the blue team (top) and for the red team (bottom) when the blue team plays `A1_M1_D3`, for each possible red play. This figure presents the same data as Figure B.3, but the y-axis is zoomed in.



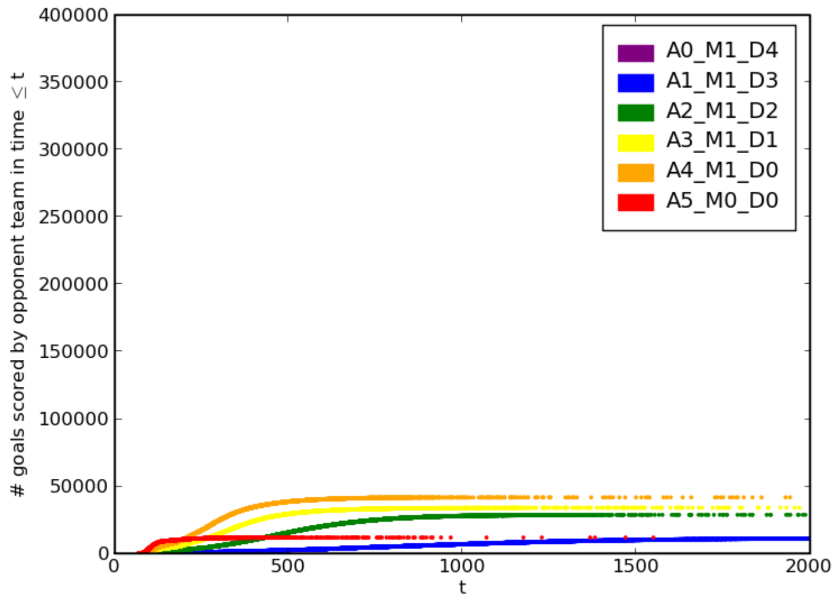
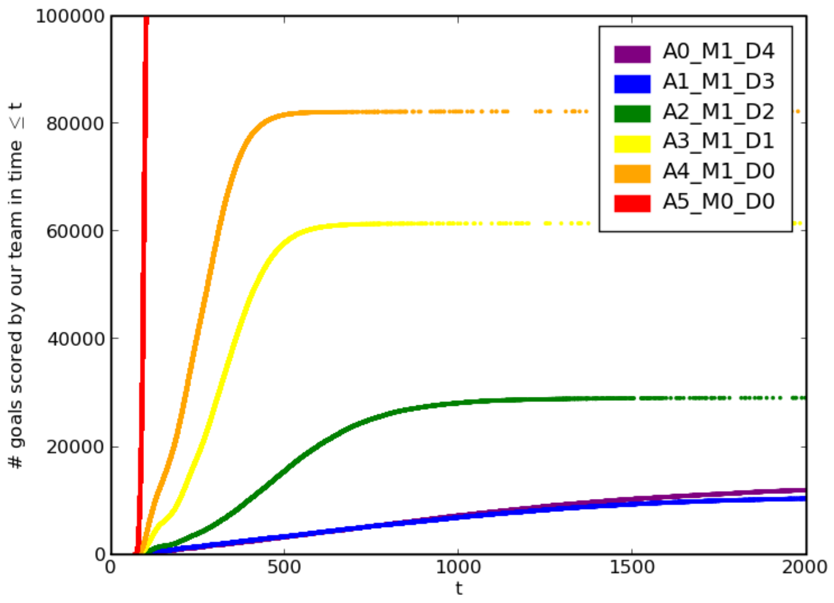


Figure B.5: Time-to-score distributions for the blue team (top) and for the red team (bottom) when the blue team plays `A2_M1_D2`, for each possible red play.



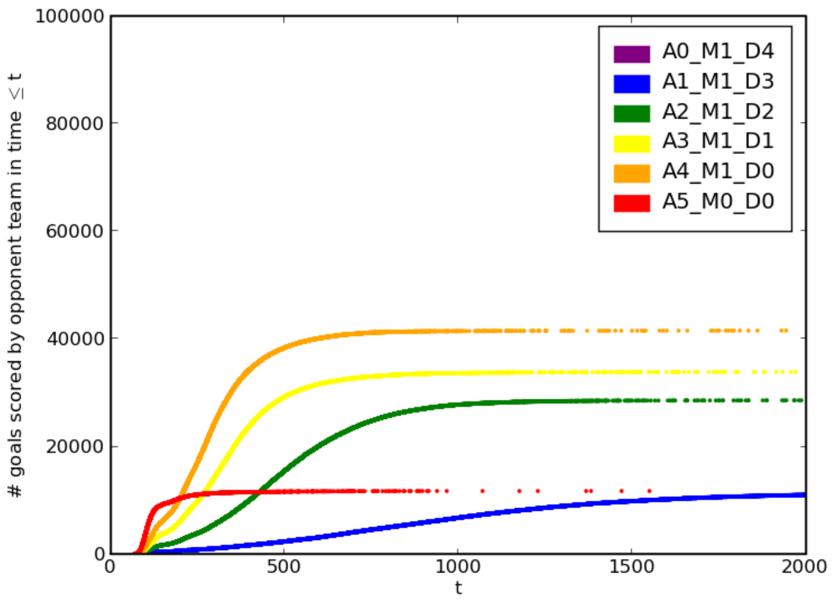
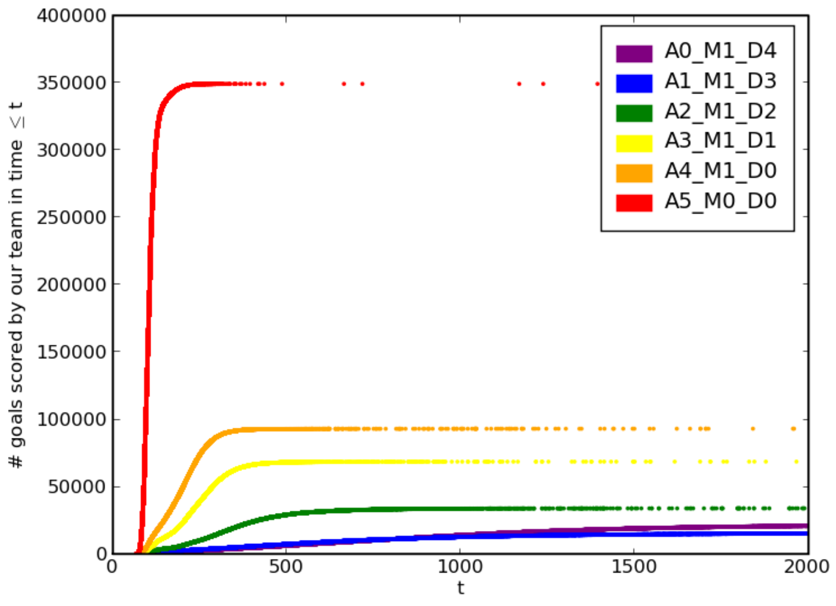


Figure B.6: Time-to-score distributions for the blue team (top) and for the red team (bottom) when the blue team plays `A2_M1_D2`, for each possible red play. This figure presents the same data as Figure B.5, but the y-axis is zoomed in.



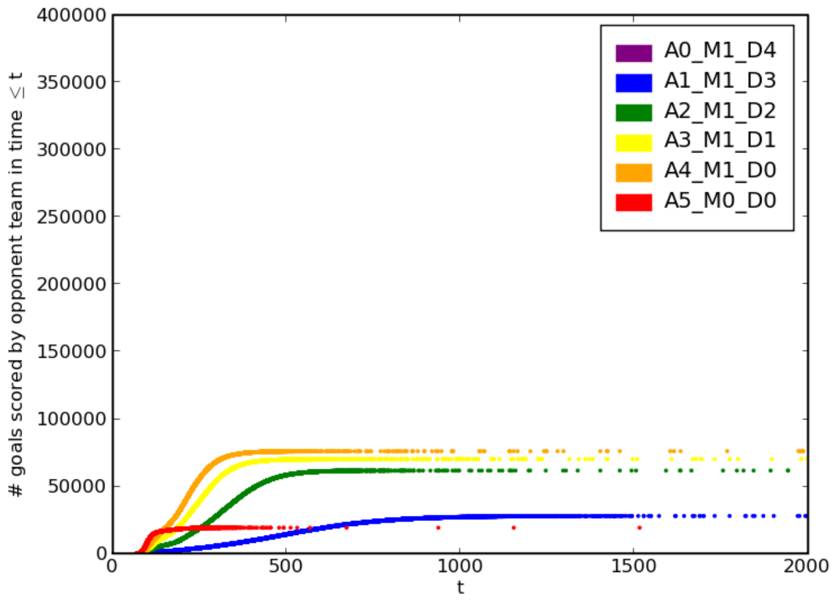
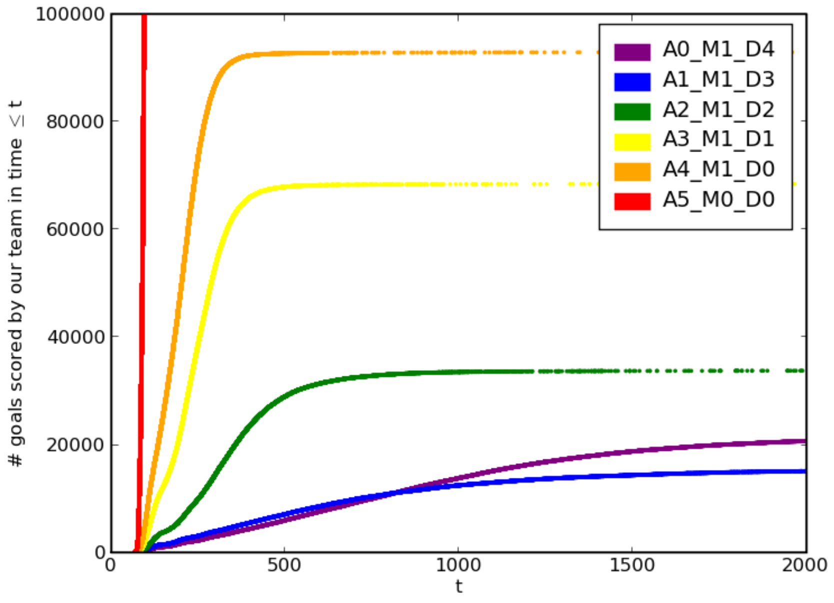


Figure B.7: Time-to-score distributions for the blue team (top) and for the red team (bottom) when the blue team plays `A3_M1_D1`, for each possible red play.



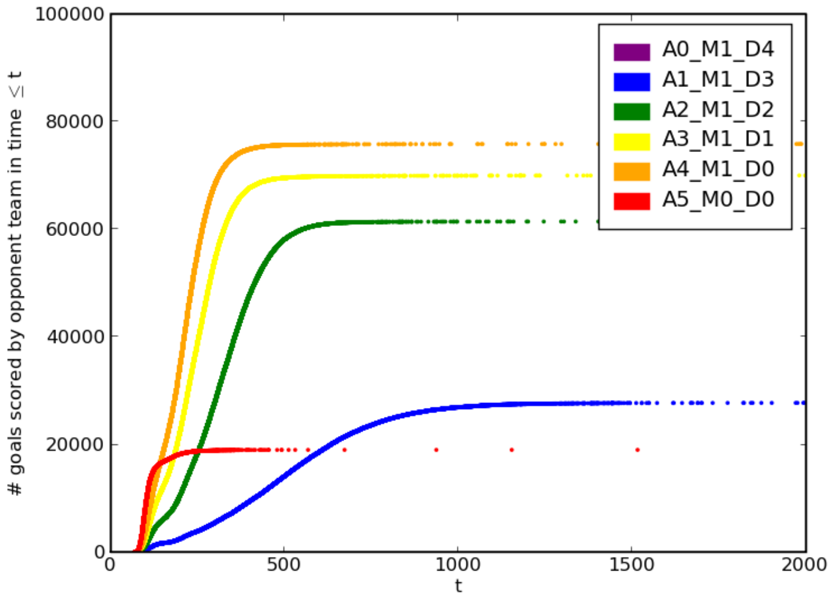
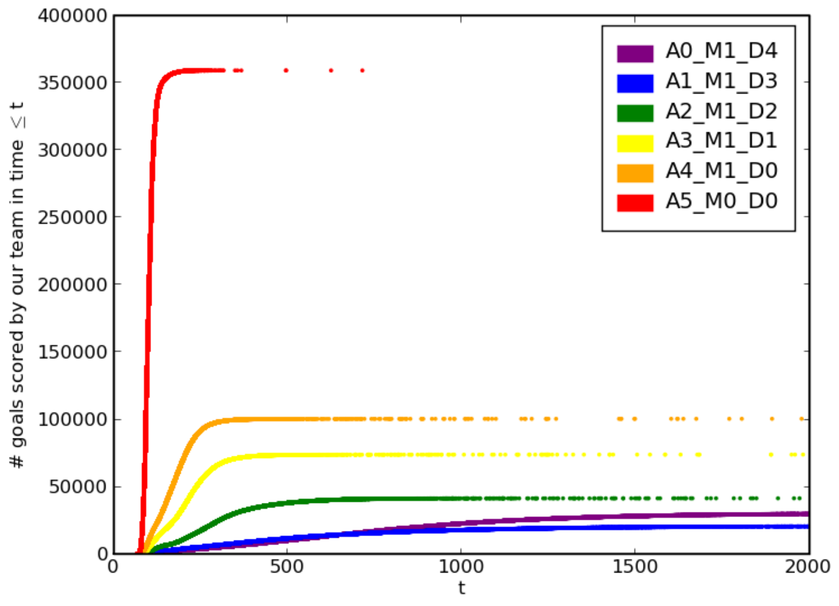


Figure B.8: Time-to-score distributions for the blue team (top) and for the red team (bottom) when the blue team plays `A3_M1_D1`, for each possible red play. This figure presents the same data as Figure B.7, but the y-axis is zoomed in.



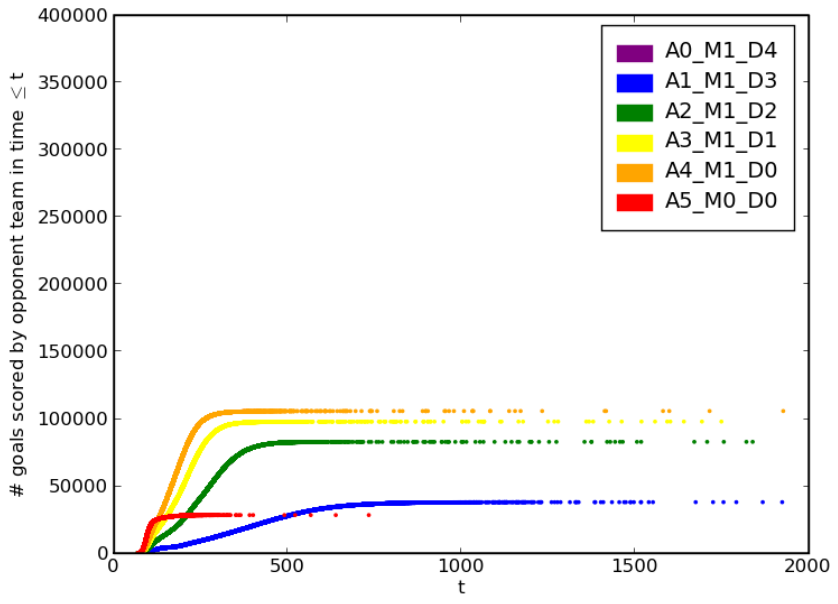
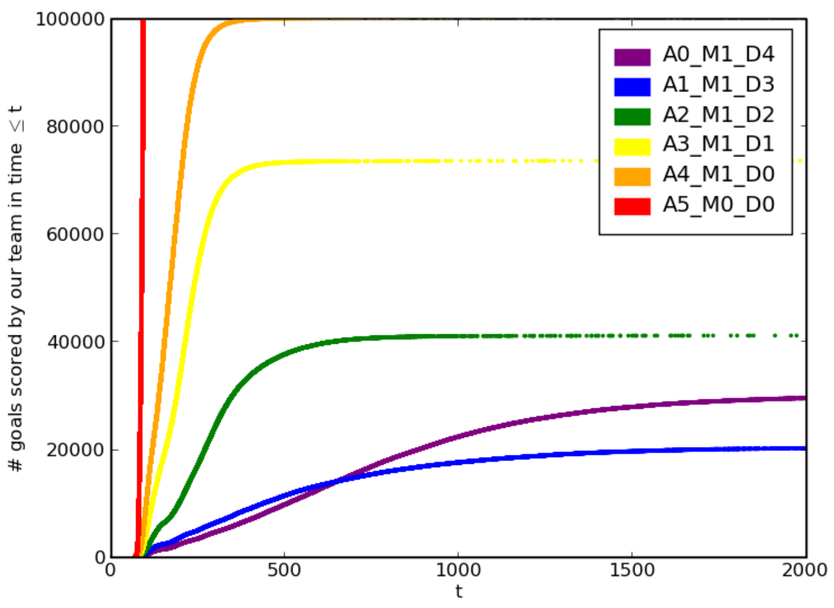


Figure B.9: Time-to-score distributions for the blue team (top) and for the red team (bottom) when the blue team plays `A4_M1_D0`, for each possible red play.



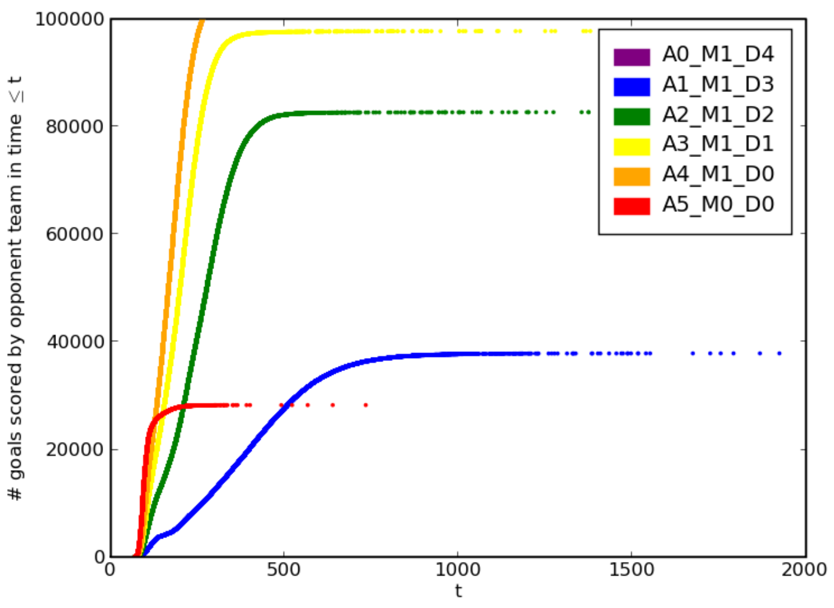
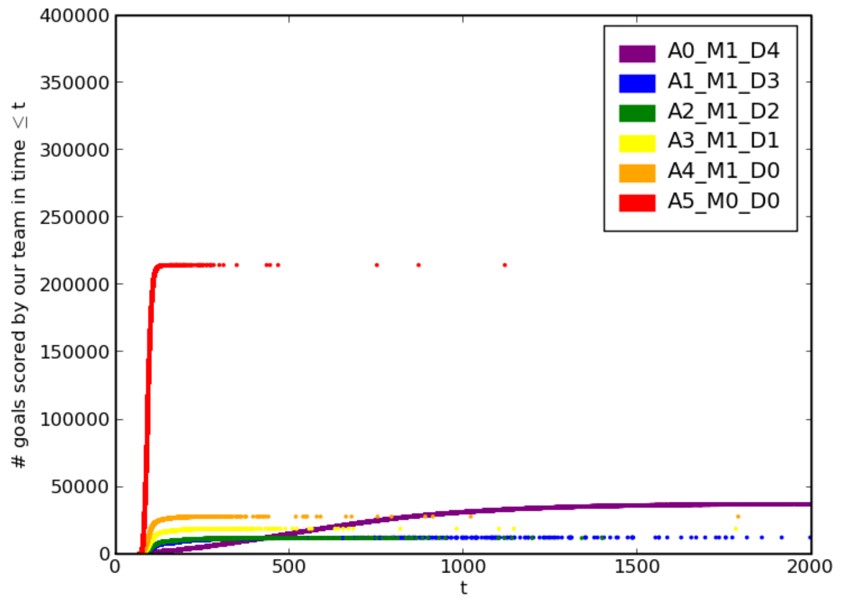


Figure B.10: Time-to-score distributions for the blue team (top) and for the red team (bottom) when the blue team plays `A4_M1_D0`, for each possible red play. This figure presents the same data as Figure B.9, but the y-axis is zoomed in.



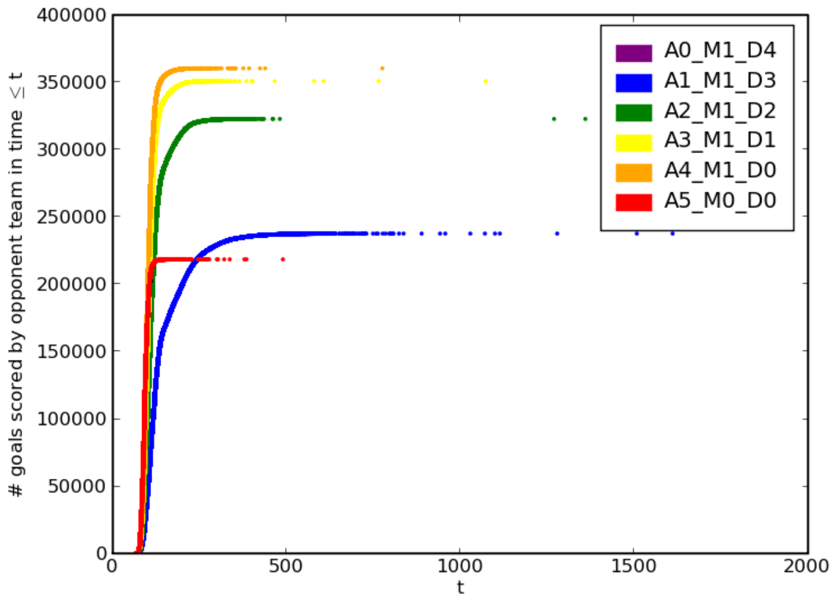
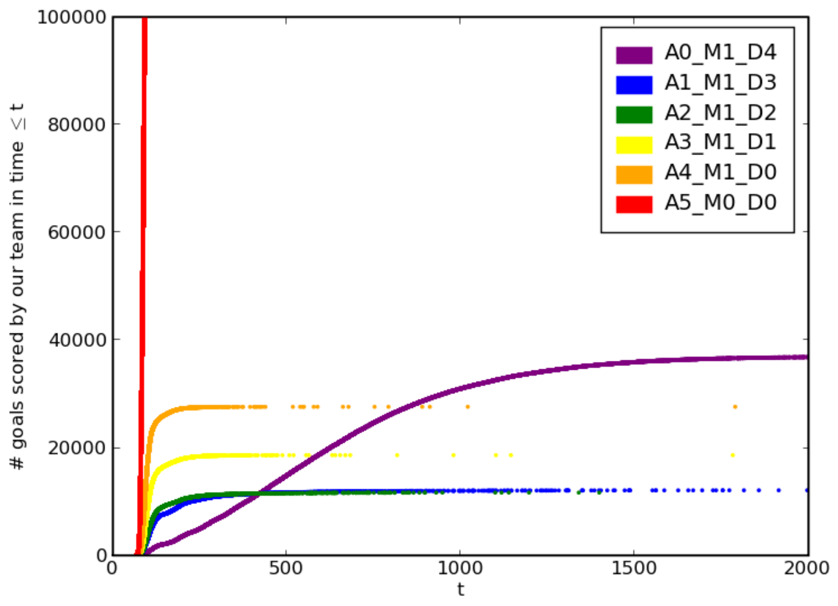


Figure B.11: Time-to-score distributions for the blue team (top) and for the red team (bottom) when the blue team plays `A5_M0_D0`, for each possible red play.



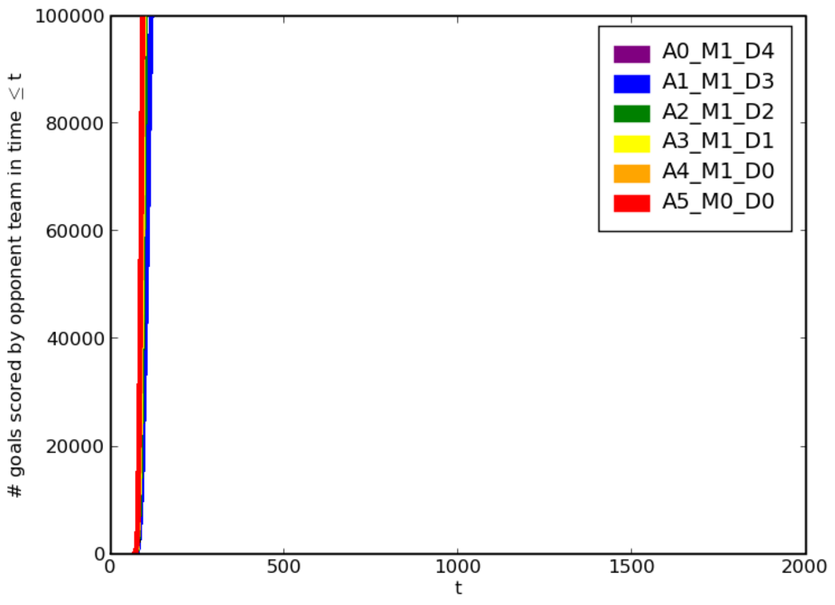


Figure B.12: Time-to-score distributions for the blue team (top) and for the red team (bottom) when the blue team plays `A5_M0_D0`, for each possible red play. This figure presents the same data as Figure B.11, but the y-axis is zoomed in.

B.3 Optimal Policies

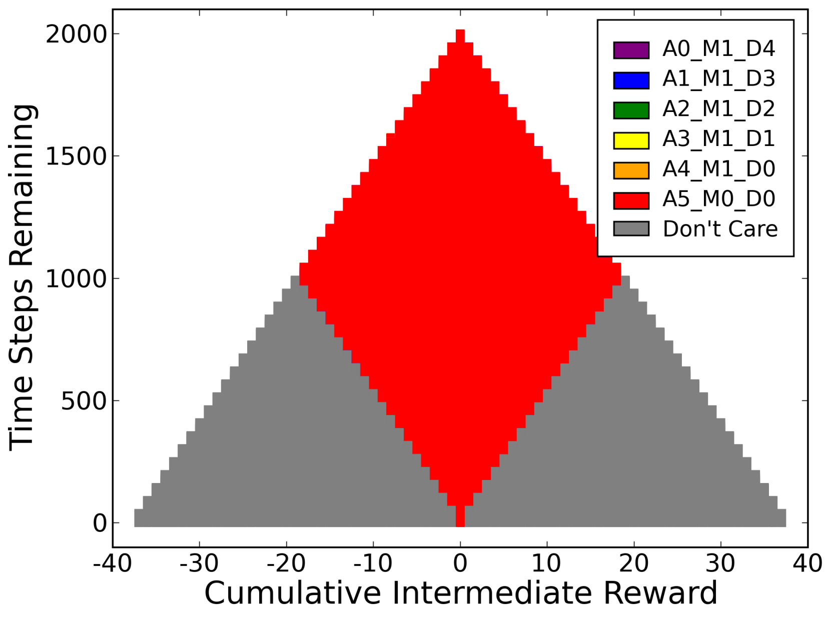


Figure B.13: The optimal policy for the CTF domain, assuming that the opponent plays A0_M1_D4 for the entire game. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference).

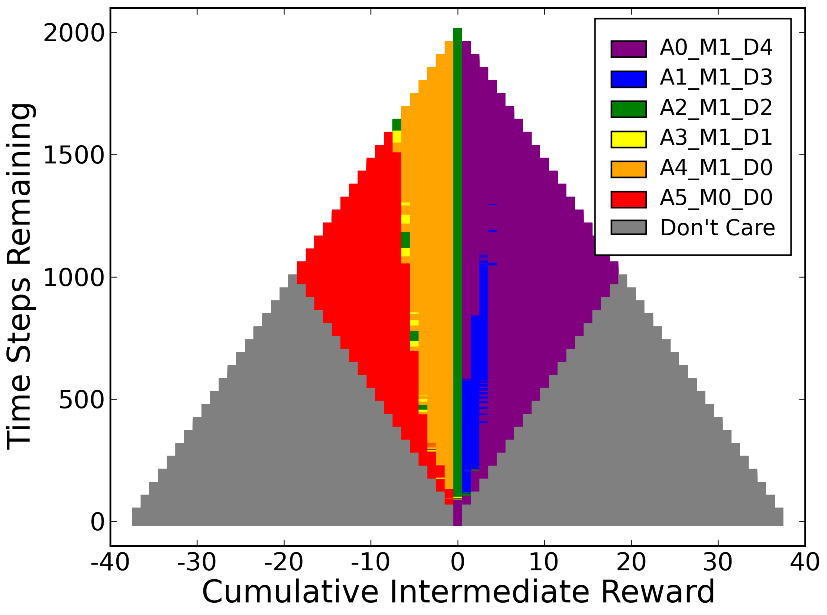


Figure B.14: The optimal policy for the CTF domain, assuming that the opponent plays `A1_M1_D3` for the entire game. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference).

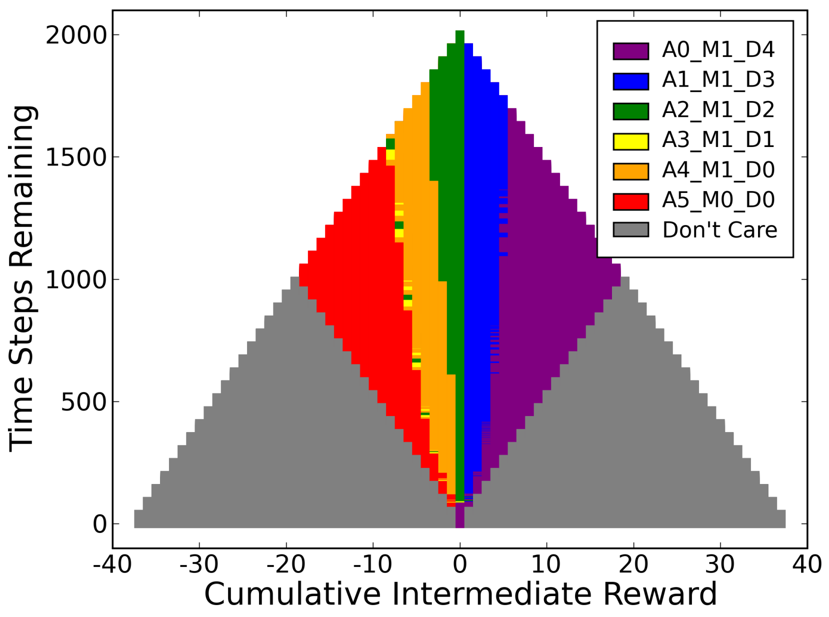


Figure B.15: The optimal policy for the CTF domain, assuming that the opponent plays `A2_M1_D2` for the entire game. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference).

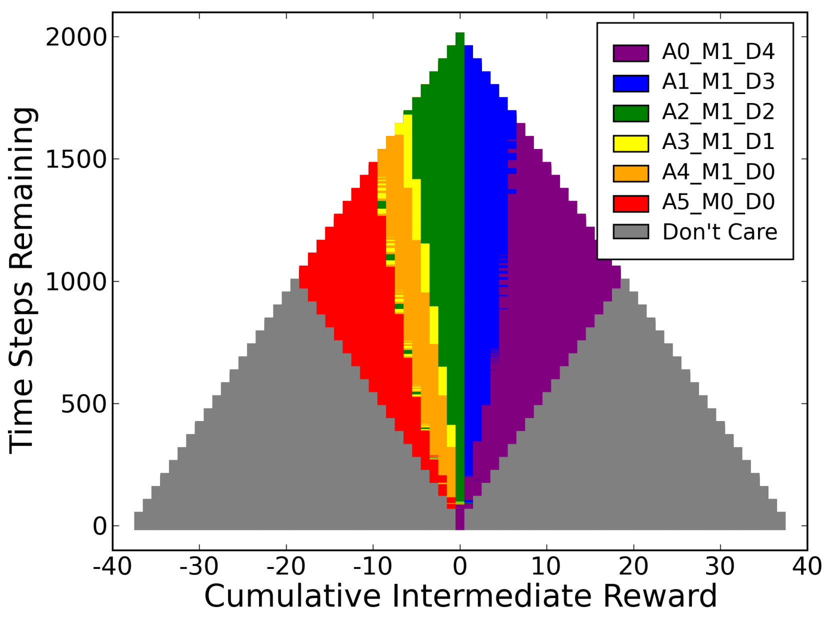


Figure B.16: The optimal policy for the CTF domain, assuming that the opponent plays `A3_M1_D1` for the entire game. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference).

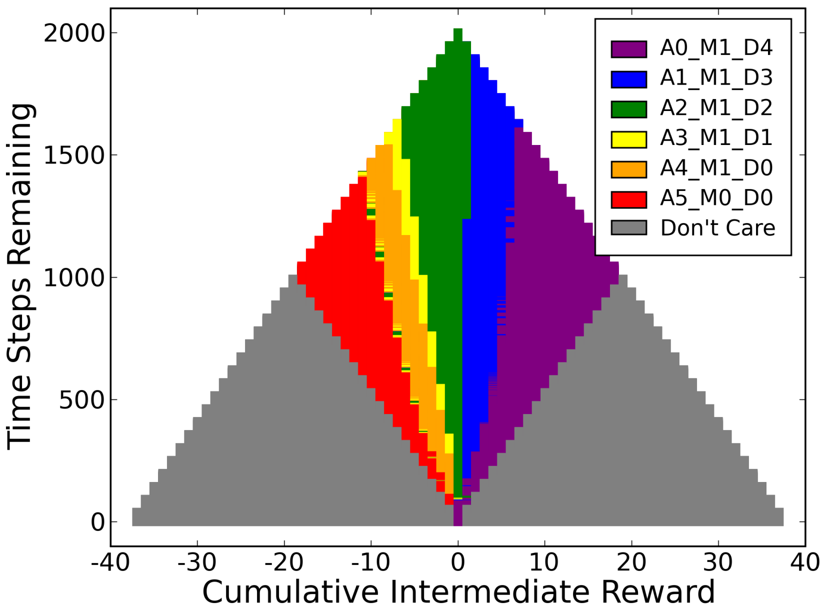


Figure B.17: The optimal policy for the CTF domain, assuming that the opponent plays `A4_M1_D0` for the entire game. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference).

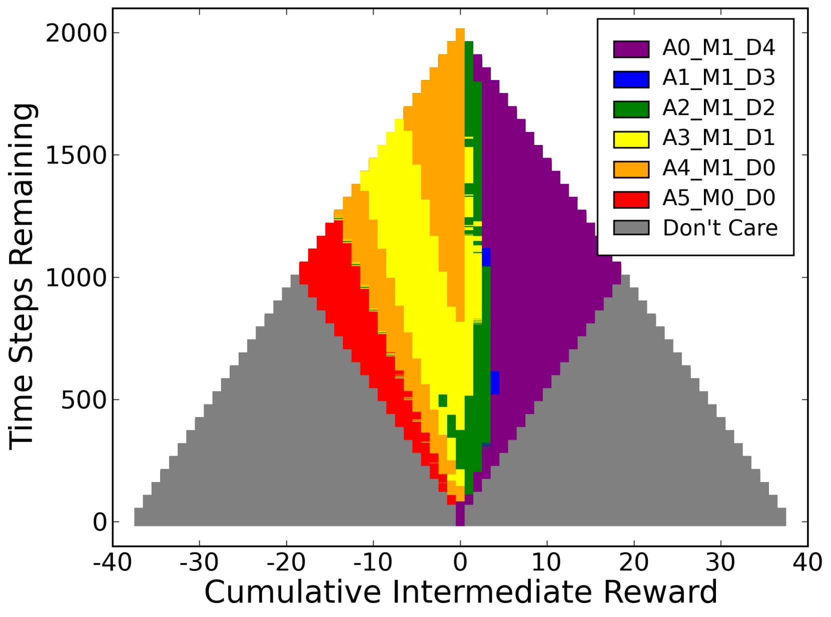


Figure B.18: The optimal policy for the CTF domain, assuming that the opponent plays `A5_M0_D0` for the entire game. The y-axis shows the number of time steps remaining; the x-axis shows the cumulative intermediate reward (score difference).

Bibliography

[1] F. Bacchus, C. Boutilier, and A. Grove. Structured solution methods for non-Markovian decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997.

[2] Fahiem Bacchus, Craig Boutilier, and Adam Grove. Rewarding behaviors. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1160–1167, Portland, Oregon, USA, 1996. AAAI Press / The MIT Press.

[3] Tucker Balch and Lynne Parker. *Robot Teams: From Diversity to Polymorphism*. AK Peters, 2002.

[4] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[5] Michael Bowling, Brett Browning, Allen Chang, and Manuela Veloso. Plays as team plans for coordination and adaptation. In D. Polani, B. Browning, A. Bonarini, and K. Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Computer Science*, pages 686–693. Springer Verlag, Berlin, Germany, 2004.

[6] Michael Bowling, Brett Browning, and Manuela Veloso. Plays as team plans for coordination and adaptation. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, June 2004.

[7] Steven J. Bradtke and Michael O. Duff. Reinforcement learning methods for continuous-time markov decision problems. In *Advances in Neural Information Processing Systems*, pages 393–400. MIT Press, 1995.

[8] Brett Browning, James Bruce, Michael Bowling, and Manuela Veloso. STP: Skills, tactics and plays for multi-robot control in adversarial environments. *IEEE Journal of Control and Systems Engineering*, 219:33–52, 2005.

[9] Frank Broz, Illah Nourbakhsh, and Reid Simmons. Planning for human-robot interaction using time-state aggregated POMDPs. In *Proceedings of the Twenty-Third Conference on Artificial Intelligence (AAAI-08)*, July 2008.

[10] G. Dudek, M. Jenkin, and E. Milios. A taxonomy of multi-robot systems. In Tucker Balch and Lynne Parker, editors, *Robot Teams: From Diversity to Polymorphism.* AK Peters, 2002.

[11] F. Dylla, A. Ferrein, G. Lakemeyer, J. Murray, O. Obst, T. Röfer, F. Stolzenburg, U. Visser, and T. Wagner. Towards a league-independent qualitative soccer theory for RoboCup. In *RoboCup 2004: Robot Soccer World Cup VIII*, Lecture Notes in Computer Science. Springer Verlag, Berlin, Germany, 2005.

[12] Frank Dylla, Alexander Ferrein, Gerhard Lakemeyer, Jan Murray, Oliver Obst, Thomas Röfer, Stefan Schiffer, Frieder Stolzenburg, Ubbo Visser, and Thomas Wagner. *Computers in Sport*, chapter Approaching a Formal Soccer Theory from the Behavior Specification in Robotic Soccer, pages 161–186. Bioengineering. WIT Press, 2008. ISBN 978-1-84564-064-4.

[13] Juan Fasola and Manuela Veloso. Real-time object detection using segmented and grayscale images. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation (ICRA 2006)*, May 2006.

[14] Brian P. Gerkey. *On Multi-Robot Task Allocation.* PhD thesis, University of Southern California, 2003.

[15] Brian P. Gerkey and Maja J Mataric. A formal analysis and taxonomy of task allocation in multi-robot systems. *Intl. Journal of Robotics Research*, 23(9):939–954, Sep 2004.

[16] Brian P. Gerkey and Maja J. Mataric. On role allocation in RoboCup. In D. Polani, B. Browning, A. Bonarini, and K. Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Computer Science*, pages 43–53. Springer Verlag, Berlin, Germany, 2004.

[17] Kwun Han and Manuela Veloso. Automated robot behavior recognition applied to robotic soccer. In *Robotics Research: the Ninth International Symposium*, pages 199–204. Springer-Verlag, 2000.

[18] T.H. Ho, C. Camerer, and K. Weigelt. Iterated dominance and iterated best response in experimental "p-beauty contests". *American Economic Review*, pages 947–969, 1998.

[19] Ronald A. Howard. *Dynamic Programming and Markov Processes.* MIT Press, 1960.

[20] M. Ani Hsieh, Anthony Cowley, James F. Keller, Luiz Chaimowicz, Ben Grocholsky, Vijay Kumar, Camillo J. Taylor, Yoichiro Endo, Ronald C. Arkin, Boyoon Jung, Denis F. Wolf, Gaurav S. Sukhatme, and Douglas C. MacKenzie. Adaptive teams of autonomous aerial and ground robots for situational awareness: Field reports. *Journal of Field Robotics*, 24(11-12):991–1014, 2007.

[21] Leslie P. Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable domains. *Artificial Intelligence*, 1998.

[22] Leslie P. Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 1996.

[23] Leslie Pack Kaelbling. *Learning in Embedded Systems.* MIT Press, 1993.

[24] H. Kitano, M. Fujita, S. Zrehen, and K. Kageyama. Sony legged robot for RoboCup challenge. In *1998 IEEE International Conference on Robotics and Automation*, volume 3, 1998.

[25] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The robot World Cup initiative. In *Proc. of the First Intl. Conf. on Autonomous Agents (Agents '97)*, 1997.

[26] William S. Krasker. Football commentary: Dynamic programming model. `http://www.footballcommentary.com/dynamicprogramming.htm`, 2004.

[27] Tim Laue and Thomas Röfer. A behavior architecture for autonomous mobile robots based on potential fields. In *8th Intl. Workshop on RoboCup 2004, Lecture Notes in Artificial Intelligence*, Lecture Notes in Computer Science, Berlin, Germany, 2004. Springer Verlag.

[28] Lihong Li, Thomas J. Walsh, and Michael L. Littman. Towards a unified theory of state abstraction for MDPs. In *Ninth International Symposium on Artificial Intelligence and Mathematics*, 2006.

[29] S.A. Lippman. Semi-Markov decision processes with unbounded rewards. *Management Science*, pages 717–731, 1973.

[30] Yaxin Liu and Sven Koenig. Existence and finiteness conditions for risk-sensitive planning: Results and conjectures. In *Proceedings of Uncertainty in Artificial Intelligence*, 2005.

[31] Yaxin Liu and Sven Koenig. Functional value iteration for decision-theoretic planning with general utility functions. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06)*, July 2006.

[32] Yaxin Liu and Sven Koenig. An exact algorithm for solving MDPs under risk-sensitive planning objectives with one-switch utility functions. In *AAMAS '08: Proceedings of the 7th International Joint Conference on Autonomous agents and Multiagent Systems*, pages 453–460, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[33] S. Mahadevan, N. Marchalleck, T.K. Das, and A. Gosavi. Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *Proceedings of the Fourteenth International Conference on Machine Learning*, 1997.

[34] Sridhar Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1-3):159–195, 1996.

[35] Sridhar Mahadevan. Optimality criteria in reinforcement learning. In *Proceedings of the AAAI Fall Symposium on Learning Complex Behaviors in Adaptive Intelligent Systems*, 1996.

[36] Colin McMillen. Capture the flag simulator source code. `http://colinm.org/thesis`, 2009.

[37] Colin McMillen, Paul Rybski, and Manuela Veloso. Levels of multi-robot coordination for dynamic environments. In *Multi-Robot Systems: From Swarms to Intelligent Automata, Volume III*, pages 53–64. Kluwer Academic Publishers, 2005.

[38] Colin McMillen and Manuela Veloso. Distributed, play-based coordination for robot teams in dynamic environments. In *RoboCup 2006: Robot Soccer World Cup X*, June 2006.

[39] Colin McMillen and Manuela Veloso. Distributed, play-based role assignment for robot teams in dynamic environments. In *Proceedings of Distributed Autonomous Robotic Systems*, July 2006.

[40] C. Miller, H. Funk, P. Wu, R. Goldman, J. Meisner, and M. Chapman. The playbook approach to adaptive automation. In *Proc. Human Factors and Ergonomics Society*, 2005.

[41] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.

[42] Martin Mundhenk, Judy Goldsmith, Christopher Lusena, and Eric Allender. Complexity of finite-horizon Markov decision process problems. *Journal of the ACM*, 47(4):681–720, 2000.

[43] Lynne E. Parker and Brad Emmons. Cooperative multi-robot observation of multiple moving targets. In *Proceedings of 1997 International Conference on Robotics and Automation*, 1997.

[44] Ronald E. Parr. *Hierarchical control and learning for Markov decision processes.* PhD thesis, UNIVERSITY of CALIFORNIA, 1998.

[45] S. D. Patek and D. P. Bertsekas. Play selection in American football: A case study in neuro-dynamic programming. *Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search: Interfaces in Computer Science and Operations Research*, page 189, 1998.

[46] R. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley, 1994.

[47] D.V. Pynadath and M. Tambe. The communicative Multiagent Team Decision Problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, 16:389–423, 2002.

[48] M.J. Quinlan and S.K. Chalup. Impact of tactical variations in the RoboCup four-legged league. In *Proceedings of the 2006 International Symposium on Practical Cognitive Agents and Robots*, pages 27–38. ACM New York, NY, USA, 2006.

[49] M.J. Quinlan, N. Henderson, R.H. Middleton, S.P. Nicklin, R. Fisher, F. Knorn, S.K. Chalup, and R. King. The 2006 NUbots team report, 2006.

[50] M.J. Quinlan, O. Obst, and S.K. Chalup. Towards autonomous strategy decisions in the RoboCup Four-Legged League. In *Proceedings of the Seventh IJCAI International Workshop on Nonmontonic Reasoning, Action and Change.*

[51] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[52] RoboCup Technical Committee. Sony four legged robot football league rule book. Available online at: `http://www.tzi.de/spl/pub/Website/History/Rules2004.pdf`, 2004.

[53] RoboCup Technical Committee. Sony four legged robot football league rule book. Available online at: `http://www.tzi.de/4legged/pub/Website/Downloads/Rules2005.pdf`, 2005.

[54] RoboCup Technical Committee. RoboCup four-legged league rule book. Available online at: `http://www.tzi.de/4legged/pub/Website/Downloads/Rules2006.pdf`, 2006.

[55] RoboCup Technical Committee. RoboCup four-legged league rule book. Available online at: `http://www.tzi.de/4legged/pub/Website/Downloads/Rules2007.pdf`, 2007.

[56] RoboCup Technical Committee. RoboCup four-legged league rule book. Available online at: `http://www.tzi.de/4legged/pub/Website/Downloads/Rules2008.pdf`, 2008.

[57] David Romer. *It's Fourth Down and what Does the Bellman Equation Say?: A Dynamic-programming Analysis of Football Strategy.* National Bureau of Economic Research, 2002.

[58] Maayan Roth, Douglas Vail, and Manuela Veloso. A real-time world model for multi-robot teams with high-latency communication. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 2494–2499, October 2003.

[59] H. Sackrowitz. Refining the point(s)-after-touchdown decision. *Chance*, 13(3):29–34, 2000.

[60] Sascha A. Stoeter, Paul E. Rybski, Kristen N. Stubbs, Colin P. McMillen, Maria Gini, Dean F. Hougen, and Nikolaos Papanikolopoulos. A robot team for surveillance tasks: Design and architecture. *Robotics and Autonomous Systems*, 40(2–3):173–183, August 2002.

[61] Peter Stone. *Layered Learning in Multi-Agent Systems.* PhD thesis, Carnegie Mellon University, December 1998.

[62] Alexander L. Strehl, Lihong Li, and Michael L. Littman. Incremental model-based learners with formal learning-time guarantees. In *Proceedings of Uncertainty in Artificial Intelligence*, 2006.

[63] Alexander L. Strehl and Michael L. Littman. An empirical evaluation of interval estimation for Markov decision processes. In *IEEE International Conference on Tools with Artificial Intelligence*, 2004.

[64] A.W. Stroupe and T. Balch. Value-based observation with robot teams (VBORT) using probabilistic techniques. In *Proceedings of International Conference on Intelligent Robots and Systems 2003*, 2003.

[65] G.S. Sukhatme, A. Dhariwal, B. Zhang, C. Oberg, B. Stauffer, and D.A. Caron. Design and development of a wireless robotic networked aquatic microbial observing system. *Environmental Engineering Science*, 24(2):205–215, 2007.

[66] G. Sukthankar and K. Sycara. Robust recognition of physical team behaviors using spatio-temporal models. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 638–645. ACM New York, NY, USA, 2006.

[67] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, 1990.

[68] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning.* MIT Press, 1998.

[69] R.S. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.

[70] Milind Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

[71] Sylvie Thiebaux, Charles Gretton, John Slaney, David Price, and Froduald Kabanza. Decision-theoretic planning with non-Markovian rewards. *Journal of Artificial Intelligence Research*, 2006.

[72] Sylvie Thiebaux, Froduald Kabanza, and John Slaney. Anytime state-based solution methods for decision processes with non-Markovian rewards. In *Proceedings of Uncertainty in Artificial Intelligence*, 2002.

[73] D.L. Vail, M.M. Veloso, and J.D. Lafferty. Conditional random fields for activity recognition. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*. ACM New York, NY, USA, 2007.

[74] Douglas Vail and Manuela Veloso. Dynamic multi-robot coordination. In *Multi-Robot Systems: From Swarms to Intelligent Automata, Volume II*, pages 87–100. Kluwer Academic Publishers, 2003.

[75] Manuela Veloso, Paul E. Rybski, Sonia Chernova, Colin McMillen, Juan Fasola, Felix von Hundelshausen, Douglas Vail, Alex Trevor, Sabine Hauert, and Raquel Ros Espinoza. CMDash'05: Team report. Available online at `http://www.cs.cmu.edu/~robosoccer/legged/reports/CMDash05-report.pdf`, 2005.

[76] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. CAPTCHA: Using hard AI problems for security. In *Eurocrypt 2003*, 2003.

[77] Luis von Ahn, Benjamin Maurer, Colin McMillen, David Abraham, and Manuel Blum. Manual character recognition using online security measures: An example of crowd computing. *Science*, pages 1465–1468, September 12, 2008.

[78] Liad Wagman and Vincent Conitzer. Strategic betting for competitive agents. In *AA-MAS '08: Proceedings of the 7th International Joint Conference on Autonomous agents and Multiagent Systems*, pages 847–854, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[79] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3–4), 1992.

[80] Thilo Weigel, Willi Auerbach, Markus Dietl, Burkhard Dümler, Jens-Steffen Gutmann, Kornel Marko, Klaus Müller, Bernhard Nebel, Boris Szerbakowski, and Maximilian Thiel. CS Freiburg: Doing the right thing in a group. *Lecture Notes in Computer Science*, 2019:52–63, 2001.

[81] M. Wiering and J. Schmidhuber. Efficient model-based exploration. In *Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior (SAB'98)*, pages 223–228, 1998.

[82] P. Zigoris, J. Siu, O. Wang, and A. Hayes. Balancing automated behavior and human control in multi-agent systems: a case study in RoboFlag. In *Proceedings of the American Control Conference*, pages 667–671, June 2003.