Norwegian University of
Science and Technology

Learning robot soccer with UCT

Vidar Holen
Audun Marøy

Master of Science in Computer Science
Submission date: June 2008
Supervisor: Helge Langseth, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

We wish to investigate the potential of reinforcement learning with UCT for learning to play simulated 2D robot soccer.

Assignment given: 14. January 2008
Supervisor: Helge Langseth, IDI

Learning to Play Robot Soccer with UCT

Vidar Holen Audun Marøy

09.06.2008

Abstract

Upper Confidence bounds applied to Trees, or UCT, has shown promise for reinforcement learning problems in different kinds of games, but most of the work has been on turn based games and single agent scenarios. In this project we test the feasibility of using UCT in an action-filled multi-agent environment, namely the RoboCup simulated soccer league. Through a series of experiments we test both low level and high level approaches. We were forced to conclude that low level approaches are infeasible, and that while high level learning is possible, cooperative multi-agent planning did not emerge.

Trondheim, June 7, 2008

Vidar Holen

Audun Marøy

Preface

This project is done as part of our fulfilment of the master program in computer science at The Norwegian University of Science and Technology (NTNU). We would like to thank the following for their help and support during the project: Our supervisor Helge Langseth, Audun's computer "Chaos" for dutifully performing all the experiments and our friends and family.

Contents

1 Introduction **1**
- 1.1 Artificial Intelligence in games . . . 1
  - 1.1.1 Entertaining AIs . . . 2
  - 1.1.2 Competitive AIs . . . 2
- 1.2 Problem statement . . . 3
- 1.3 Approach . . . 4
- 1.4 Overview . . . 4

2 Prestudy **5**
- 2.1 RoboCup . . . 5
  - 2.1.1 Introduction . . . 5
  - 2.1.2 Leagues . . . 5
- 2.2 The RoboCup 2D Simulation . . . 7
  - 2.2.1 Simulator . . . 7
  - 2.2.2 World . . . 7
  - 2.2.3 Players . . . 9
  - 2.2.4 Coaches . . . 10
- 2.3 Machine learning theory . . . 10
  - 2.3.1 MDP . . . 10
  - 2.3.2 RL . . . 13
  - 2.3.3 UCT . . . 15
  - 2.3.4 Soccer server as a POMDP . . . 17
  - 2.3.5 MDP presumptions in soccer server . . . 17
  - 2.3.6 Learning in multi-agent environments . . . 19
- 2.4 The state of the art . . . 20
  - 2.4.1 BrainStormers . . . 21
  - 2.4.2 Multi-agent reinforcement learning . . . 23
  - 2.4.3 State of the Art in contemporary soccer games . . . 24

3 Architecture **25**
- 3.1 Colugo . . . 25
  - 3.1.1 Overview . . . 25
  - 3.1.2 The UCT learning component . . . 26

3.1.3 The modified BrainStormers client 27

4 Experiments 29
4.1 Introduction 29
4.1.1 Experiment infrastructure 29
4.2 Experiment 1 – Translating vision directly into states 30
4.2.1 Setting 30
4.2.2 States 30
4.2.3 Actions 30
4.2.4 Weaknesses 30
4.2.5 Expected results 30
4.3 Experiment 1 - Results 31
4.3.1 Training results 31
4.3.2 Problems with the experiment 31
4.3.3 Conclusion 31
4.4 Experiment 2 – Triangulated position 31
4.4.1 Setting 31
4.4.2 States 31
4.4.3 Actions 32
4.4.4 Weaknesses 32
4.4.5 Expected results 32
4.5 Experiment 2 - Results 32
4.5.1 Training results 32
4.5.2 Problems with the experiment 33
4.5.3 Conclusion 33
4.6 Experiment 3 – Using the BrainStormers framework 33
4.6.1 Setting 33
4.6.2 States 33
4.6.3 Actions 34
4.6.4 Weaknesses 35
4.6.5 Expected results 35
4.7 Experiment 3 - Results 35
4.7.1 Training results 35
4.7.2 Problems with the experiment 35
4.7.3 Conclusion 36
4.8 Experiment 4 – 2 Colugos VS 1 BrainStormer 36
4.8.1 Setting 36
4.8.2 States 37
4.8.3 Actions 37
4.8.4 Weaknesses 37
4.8.5 Expected results 37
4.9 Experiment 4 - Results 38
4.9.1 Training results 38

4.9.2 Problems with the experiment . . . 38
4.9.3 Conclusion . . . 38
4.10 Experiment 5 – 2 VS 1: take 2 . . . 39
4.10.1 Setting . . . 39
4.10.2 States . . . 39
4.10.3 Actions . . . 39
4.10.4 Weaknesses . . . 39
4.10.5 Expected results . . . 39
4.11 Experiment 5 - Results . . . 40
4.11.1 Training results . . . 40
4.11.2 Problems with the experiment . . . 40
4.11.3 Conclusion . . . 40
4.12 Experiment 6 – 2 VS 1: Forced pass-interception . . . 40
4.12.1 Setting . . . 40
4.12.2 States . . . 41
4.12.3 Actions . . . 41
4.12.4 Weaknesses . . . 41
4.12.5 Expected results . . . 41
4.13 Experiment 6 - Results . . . 41
4.13.1 Training results . . . 41
4.13.2 Problems with the experiment . . . 42
4.13.3 Conclusion . . . 42
4.14 Experiment 7 – 2 VS 2 . . . 43
4.14.1 Setting . . . 43
4.14.2 States . . . 43
4.14.3 Actions . . . 43
4.14.4 Weaknesses . . . 44
4.14.5 Expected results . . . 44
4.15 Experiment 7 - Results . . . 44
4.15.1 Training results . . . 44
4.15.2 Problems with the experiment . . . 44
4.15.3 Conclusion . . . 47

5 Results **49**
5.1 Introduction . . . 49
5.2 Summary . . . 49

6 Software Evaluation **51**
6.1 Soccer Server and related tools . . . 51
6.2 BrainStormers . . . 51
6.3 Kerodon UCT base . . . 52

7 Project evaluation, conclusions and further work 53
7.1 Project Evaluation 53
7.2 Conclusions 53
7.2.1 Answer to problem statement 53
7.3 Further work 54
7.3.1 Player-based or zone-based defence 54
7.3.2 Colugo agent vs BrainStormers agent: not a fair match 54
References 57

Chapter 1

Introduction

This chapter gives an introduction to the project. Section 1.1 presents AI used in computer games. Section 1.2 defines the problem statement for the project. Section 1.3 presents the approach we will use to answer the problem statement and Section 1.4 gives an overview of the organisation of the rest of the report.

1.1 Artificial Intelligence in games

AI in games is the parts of a game that make game elements behave in a seemingly intelligent way. It is related to, and often uses techniques from computer science-AI, but also includes elements like scripts and finite state machines that are not considered part of the academic field of AI.

Some of the first computer games were experiments in AI. Programs for playing checkers and chess were developed in the early 50's, and were capable of beating inexperienced but adult humans. However, the programs ran on extremely expensive room-sized computers and were therefore not available for the general public.

When the first commercial computer games were developed in the 60's and early 70's, they did not feature computer opponents at all. They just pitted one human player against another, and consequently didn't need any AI elements. In the 70's games with computer controlled enemies started to appear and thus there was a need for AI elements controlling these enemies. In the beginning they were controlled by static scripts that soon were expanded to incorporate random elements. This technique predominated for a long time, and scripted behaviour is still an important part of game AI.

As new game genres emerged, the AI had to handle new tasks, and more sophisticated AI techniques had to be developed. Real time strategy (RTS) games introduced the problems of many objects, incomplete information, path-finding problems, real-time decision making and economic planning, among other things (Schwab, 2004). Finite state machines were utilised to handle many of these problems, as they enabled the game to choose between different scripted strategies based on the current state of the game. Later games have also used computer science-AI techniques like neural networks and reinforcement learning to create evolve the intelligent behaviour in the game based

Figure 1.1: Counter-strike, a first-person shooter where a computer's instantaneous reflexes gives it the upper hand

on experience gained from playing against the human player.

Game AI can be divided into two categories based on its objective, entertaining AI and competitive AI.

1.1.1 Entertaining AIs

Entertaining AI are the agents whose goal is to entertain the player. In some games, the computer has severe advantages or disadvantages over human players, and human players vary in skill level, but the game should remain balanced to keep the player's attention.

A computer player has significant advantages in games of speed and accuracy. In First Person Shooters, such as Counter-strike in Figure 1.1, a computer will use fractions of a millisecond to perfectly aim and fire at the opponent's head, and in puzzle games, the computer can think many moves ahead even when the game runs too quickly for a human to follow. A human will quickly tire of being beaten, so the AI might miss on purpose or delay its reflexes for a more even match.

Similarly, computers have trouble with games with many variables such as Real-Time Strategy games. In these cases, a human will win too easily, so the computer might cheat by doubling its resources, increasing dealt damage or using information from the game engine that is inaccessible for the human like the positions of all enemy units. This will make the game more fun for the human, though it is not a valid strategy from an academic point of view.

To accommodate different player skills, games will either provide difficulty settings to determine how much to cheat or fumble, or use so-called rubberbanding, where the AI will adjust its skill level during the game depending on whether the player is currently in the lead or not.

1.1.2 Competitive AIs

The goal of a competitive AI agent is to win the game or achieve the highest possible score within the game rules, as a competition between human and computer, between

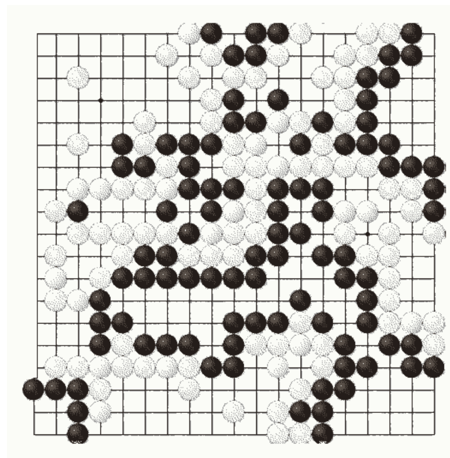

Figure 1.2: The game of Go, made computationally difficult by a large state space. The computer, like a beginning player, can start with stones to even the game against experienced opponents.

programmers, or simply to produce a good result in non-game settings. This is the most common form of AI in games with clearly defined rules, such as board games. The most prominent example of competitive AI is the chess tournament between chess champion Garry Kasparov and IBM's chess computer Deep Blue (Schaeffer & Plaat, 1997).

In board games and similar, competitive AI and entertaining AI is often the same thing. One side can be awarded a handicap according to the standard game rules designed for balancing human games, such as "knight odds" in chess, different time limits in fast chess, additional stones in Go (Figure 1.2) or extra points in bridge. However, often the point is to face an opponent on equal footing to determine the better player.

AI competitions have been around for nearly as long as game AI itself, from the first chess tournament between an American and a Soviet chess program in 1966 (played by telegraph over the course of nine months and ending with the soviet computer KAISSA winning), to current games and annual competitions like Core Wars, Robocode, the Computer Olympiad and RoboCup.

1.2 Problem statement

Currently, the most successful team in the RoboCup simulation 2D league (see Section 2.1.2, BrainStormers (Riedmiller & Gabel, 2007), uses Reinforcement Learning (RL) to develop most of the different parts of its player-controlling agents. They have decomposed the problem into a hierarchy of smaller problems, and use either RL or manual scripting to solve these problems, depending on what produces the best results. They also use RL to find the best ways of combining the different parts, and for developing strategies for the whole team. One of the reasons why they choose to decompose the task is that their RL algorithms can not handle the complexity and large state spaces

that would occur if you were to solve the entire problem at once. Even some of the problems they have after decomposition, such as getting all 11 players to act together are too complex to use RL to solve them. This leads to rigid and conventional soccer tactics, and reduces the independence and work-to-result ratio of the agent. We wish to examine the potential for more holistic approaches in developing RoboCup agents.

Using the UCT (Kocsis & Szepesvári, 2006) algorithm when choosing actions in RL has previously shown good results in complex environments with a large state spaces, for example learning to play the game Go (Gelly, Wang, Munos, & Teytaud, 2006) and developing strategies in RTS games (Holen & Marøy, 2007).

If RL could be used directly, or with less problem decomposition than what BrainStormers do, you would introduce less bias towards a certain team structure or certain tactics to employ. It would also require less expert knowledge about the environment to learn in. Because of this, we would like to investigate if using the UCT algorithm enables our agents to directly learn how to play robot soccer without splitting the task into several subtasks. This leads to the following problem statement:

Problem Statement: *Is it possible to use reinforcement learning with UCT to directly learn how to play robot soccer?*

1.3 Approach

To answer the problem statement we will employ the following approach:

1. Develop and implement a learning module that interacts with the soccer server software
2. Perform learning experiments with increasing complexity
3. Evaluate the results of our experiments

1.4 Overview

Chapter 2 describes RoboCup and the simulation domain, machine learning theory as applicable to the project, and the current state of the art of RoboCup and multi-agent reinforcement learning. Chapter 3 details the design of our agent system. Chapter 4 describes all experiments and their results, while the overall final results are summarised in Chapter 5. In Chapter 6 we evaluate the software used during the experiment, and the report concludes with Chapter 7, which evaluates the project and presents conclusions and ideas for further work.

Chapter 2

Prestudy

This section will present theoretical and historical background information necessary for the project. It will also present and justify some of the design decisions we make. Section 2.1 presents the RoboCup Challenge, Section 2.2 describes the simulation software we will use, Section 2.3 gives an introduction to relevant machine learning theory and Section 2.4 presents some examples of state of the art RoboCup agents and other multi agent reinforcement learning projects.

2.1 RoboCup

This section introduces RoboCup and its different leagues.

2.1.1 Introduction

RoboCup is an international robotics competition held every year since 1997. It was originally devoted entirely to soccer, but since 2001 it also includes robotic rescue leagues and a set of junior leagues. This section focuses on the soccer leagues.

The vision of RoboCup is to "develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team by 2050". This goal is to be achieved by providing a set of standard problems for rating teams and techniques for all facets of the game.

2.1.2 Leagues

RoboCup has competitions in five different soccer leagues, focusing on different aspects of robotics.

Simulation league

The simulation league is played by a team of eleven independent players in a simulated 2D or 3D world. The players are given sensory information similar to that a mechanical

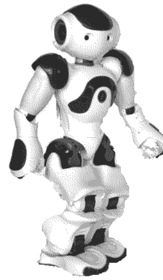

Figure 2.1: Aldebaran Nao^{TM}, the hardware base for the RoboCup Standard Platform League

robot would receive, namely fuzzy information about things in its current field of view as well as some rudimentary communication through simulated shouted phrases.

The league focuses on software development and agent co-operation, and is played on an official simulation server known as “soccerserver”, which is freely available. The clients communicate with the server through UDP packets, and can get sensory information and perform actions in discrete time steps, usually every 100ms. It’s trivial to cheat by providing additional means of error free communication between agents, so the rule about having independent players is a gentleman’s agreement.

Small Size league

The small size league consists of teams of five robots that can fit inside a 18cm circle, and allows an overhead camera for sensing and an external computer for controlling the agents directly via radio. Robots have to be marked visually so that both teams can track each others players. The agents are not required to be autonomous.

Middle Size league

The middle size league is played by six robots between 30–50cm of width, with all sensors on-board. The robots can communicate as they wish through a supplied WLAN, even with off-board computers, as long as there is no human interaction.

Standard Platform league

The standard platform league is played with a predefined robot. The chosen robot was up until 2007 the dog-like Sony Aibo, but from 2008 the standard platform league will use humanoid Aldebaran Nao robots seen in Figure 2.1. This league is a cross between the simulation league and the middle-size league in the sense that it’s a physical game, but mechanics skills are not necessary.

Humanoid league

The humanoid league was introduced in 2002, and is played by teams of two humanoid, bipedal robots, in one of two different size classes (under 60cm and under 130cm). They are encouraged to use sensors analogous to humans, placed in the same positions on the body. They can communicate with each other over a supplied WLAN, but not to external computers, and they have to be able to play even without WLAN communication. This is the league that is most directly associated with the RoboCup goal of super-human players.

2.2 The RoboCup 2D Simulation

This section describes the simulation domain of the RoboCup 2D league.

2.2.1 Simulator

Games are hosted and played on a RoboCup soccer server, an official software package freely available for Unix-like systems. The server accepts network connections from each agent (11 agents per team), from which it receives commands and sends sensor input. Agent software can take any form, and runs on the team's own equipment during matches.

Humans can watch the match by connecting a monitor application to the server. The standard monitor is a simple symbolic bird's eye view of the soccer field, but monitors that show the game as played by 3D humanoids are also available.

2.2.2 World

This section describes the simulated world.

Play field

The simulated world consists of a single soccer field, around 100m $\times$ 70m, plus five meters of space beyond the sideline in all directions. Visually distinct flags are placed on and around the field at specific positions, and are used by players for navigation (in lieu of painted lines). The flags do not affect other objects; players can run straight through them. A soccer field with the on-field flags marked is shown in Figure 2.3. The play field is entirely flat, and objects can not pass over or under each other.

Physical properties

The world includes inertia, friction, noise and wind which affects the movement of objects, and sensor fuzziness and limited visibility/audibility which affects sensor input.

Inertia affects players both when turning and when running. The player can easily turn 90° when standing still, but if running at full speed, he can not turn more than 30°. Friction applies to the players and the ball, so dashes and kicks will quickly dissipate.

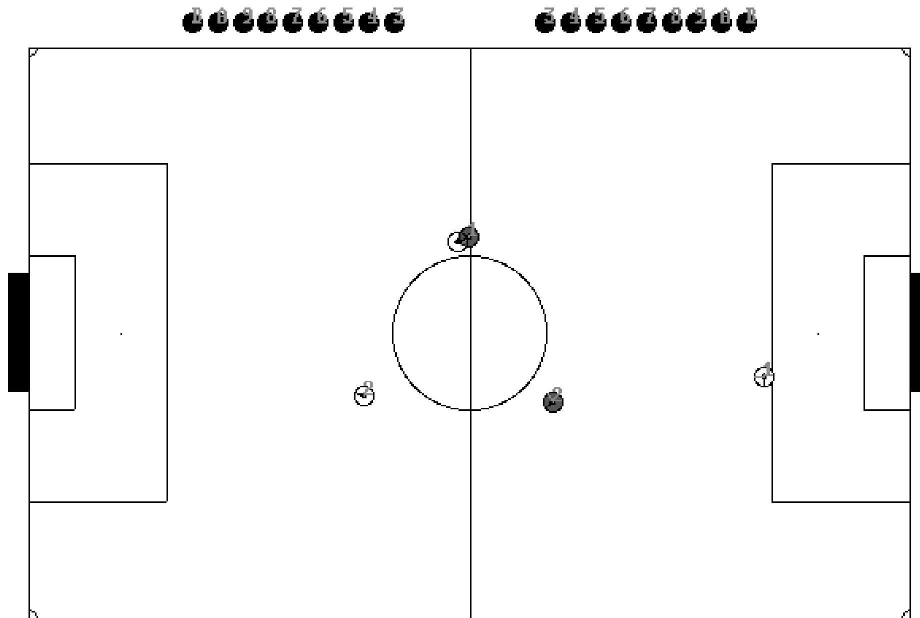

Figure 2.2: Soccer Monitor showing a 2 vs 2 game

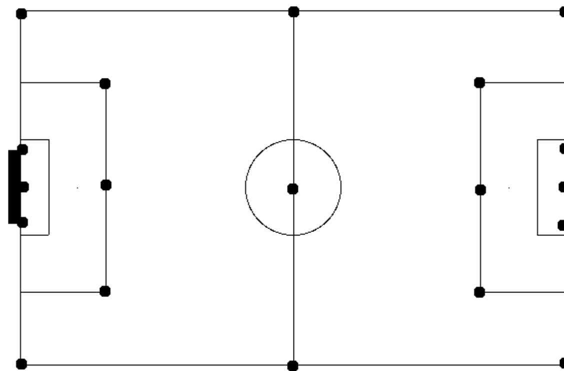

Figure 2.3: Soccer field with some flags marked

Other physical phenomenons are modelled through noise, which is applied to all objects to slightly randomise their speed and direction. Finally, wind will push the objects in a variable direction.

Sensors on agents can only see a certain distance, and the further away the object is, the less information and more uncertainty there is. This is further described in Section 2.2.3.

Time

Time is simulated in discrete cycles, ten per second by default. Sensor input includes the cycle number in which they were captured (input can be lost due to dropped network packets). Cycles progress asynchronously, so if agents use too long to calculate a move, they will lag behind.

2.2.3 Players

This section describes the players and their capabilities.

Vision

Vision is modelled as a limited computer vision system. The agent can choose if it wants a wide, normal or narrow field of view, and high or low quality data. With a wide field of view, there is more uncertainty in the input. If the quality is high, uncertainty is reduced, but update frequency is halved. The default is normal angle, low quality input, which gives one vision update every other cycle.

There are three kinds of objects the player can see: the ball, flags and other players. The amount of information varies with the distance. From furthest away to closest, the information obtained is:

- Type (player, flag or ball), direction and distance
- Player's team
- Change speed of distance and direction
- Body/head facing angle
- Jersey number

Hearing

Players, coaches and the referee can shout commands. The agent knows if it's a coach or referee that shouts the message, otherwise it can only sense the direction it comes from. The agent can only hear a certain amount of messages over a given time interval. With the default settings, this is one message every other cycle. Referee messages come across regardless.

Introspection

The agent can sense a number of things about itself:

- The current view mode (field of view, quality)
- Remaining stamina
- Current speed and angle relative to the body direction
- The current head direction relative to the body

Actions

An agent can perform a number of actions. Only one action will be performed per cycle. The list of actions are:

turn m	Turn body with moment m
dash p	Run with power p ($p < 0$ means backwards)
kick p d	Kick ball with power p in the direction d
catch d	Catch the ball coming from direction d (goalie)
turn_neck d	Turn neck d degrees in relation to the body
say m	Shout the message m
change_view w q	Change view mode to width w and quality q

2.2.4 Coaches

Each team can have a coach, which has full and error free knowledge of the players. There are two kinds of coaches, online and offline. The offline coach can move players and the ball around on the field, replenish their stamina and set the play mode as he sees fit. This can be used by the programmer to make training scenarios and test the agents. Offline coaches are obviously not used during competitions.

The online coach can only shout occasional comments from the sideline, either as freeform text or using a standardised coach language. The freeform text is not likely to be understood by the opposing players but the standardised coach language was developed to enable coaches to communicate with teams developed by different developers, and opposing players could easily be made to understand what your coach is shouting in this language.

2.3 Machine learning theory

This section will present the theory around the machine learning techniques that are relevant to this project. It is based on (Holen & Marøy, 2007).

2.3.1 MDP

This section presents Markov Decision Processes

Definition of an MDP

A Markov Decision Process is a tuple $< S, A, T, R >$, where S is a finite discrete set of environment states, A is a finite discrete set of actions available to the agent, $T : S \times A \rightarrow \Pi(S)$ is a transition function giving for each state and action a probability distribution over states, $R : S \times A \rightarrow \mathbb{R}$ is a reward function of the agent, giving the expected immediate reward received by the agent under each actions in each state (Yang & Gu, 2004). The goal of the process is to elicit an optimal policy $\pi^* : S \rightarrow A$ that for each state provides the optimal action to take to maximise the future payoff.

The future payoff R^π of following policy π can be defined in several ways, depending on the overall goal of the process. It can be the expected rewards for a finite number of future steps k:

$$R^\pi = R(s_0) + R(s_1) + ... + R(s_k) \tag{2.1}$$

It can be the average reward for all future steps:

$$R^\pi = \frac{\sum_{i=0}^{n} R(s_i)}{n} \tag{2.2}$$

or it can be the discounted cumulative reward for all future steps:

$$R^\pi = \sum_{i=0}^{n} \gamma^i R(s_i) \tag{2.3}$$

Here the reward obtained in future step number i is multiplied by γ^i, for some $\gamma \in [0, 1\rangle$. With a low γ, the agent will prefer immediate rewards, while with a γ closer to 1, it will rather take its chances on a greater reward further in the future. The discounted cumulative reward method is by far the most common due to its good convergence properties, and will be the one used in our experiments.

There are two different ways of looking at uncertainty in a MDP. The first is that T is deterministic, but that the state is only partially known and the non-observable variables cause non-deterministic output, this is called a partially observable MDP (POMDP), and is discussed further in Section 2.3.1. The second is that the current state is fully known, but T is non-deterministic and produces new states with some fixed distribution.

A state can be *absorbing*, meaning that it is an end state or goal state where the simulation stops. An absorbing state is usually associated with some large positive or negative reward to make the agent either seek or avoid that state in the future. The most direct way of defining reward functions is to only give non-zero rewards for actions leading to absorbing states, since they tend to be trivial to evaluate (either the agent reached the supervisor's goal, or it did not).

Solving an MDP

Here we will present the two most commonly applied iterative methods for solving MDPs, value iteration and policy iteration. The presentation is based on (Jensen & Nielsen, 2007).

Value iteration The value iteration algorithm starts with an initial guess $R^{\pi^*}(s)$ for the reward obtained from starting in state s and following the optimal policy π^* for the current reward distribution. The algorithm then proceeds to iteratively update $R^{\pi^*}(s)$ according the the following updating function:

$$R_{t+1}^{\pi^*}(s) = R(s) + \gamma \cdot \max_a \sum_{s'} T(s'|a,s) R_t^{\pi^*}(s') \tag{2.4}$$

Here, $R_t^{\pi^*}(s)$ denotes the reward-guess at step t and γ is the discounting variable from Equation 2.3. The updating process continues until a stopping criteria is met. This might be after a fixed number of iterations, or until the change in values is smaller than a threshold value. The algorithm can be shown to converge to the true reward distribution that we are searching for, see (Jensen & Nielsen, 2007).

Policy iteration Policy iteration starts with generating a random policy π_0. The algorithm then iteratively generates a reward function R^{π_i} based on the current policy π_i, and then updates the policy based on this new reward function. The policy is updated by the following updating function:

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s'|a,s) R^{\pi_i}(s') \tag{2.5}$$

The algorithm runs until $\pi_{i+1} = \pi_i$.

Policy algorithm is more complex to set up than value iteration, but it is generally faster to solve, as it searches a set of discrete policies opposed to the continuous space of rewards searched by value iteration. It is also easier to solve since finding R given π is a linear problem.

MDP Constraints

The most notable constraints of MDPs are that the distribution for the transition function T and the expected reward function R must be known.

POMDP

A Partially Observable Markov Decision Process is an MDP where you are unable to observe the entire state of the world. (Kaelbling, Littman, & Cassandra, 1995) To describe it as a tuple we need two extra elements. Ω is the finite set of observations the agent can make, and $O : S \times A \rightarrow \Pi(\Omega)$ is an observation function giving for each state and action a probability distribution over observations. The tuple is then: $<$

$S, A, \Omega, O, T, R >$. You no longer can be sure of which state you are in at a given time, so cannot base your decisions on the state s, but rather no all actions and observations made since you started. Luckily, this can be encoded into a belief state that is a probability distribution over states in the world. This distribution is updated with every new action taken and following observation. This continuous state space can be used as the basis of a new MDP where the actions A are the same as in the old POMDP, and the transition function T and reward function R are based on the corresponding old functions. This new MDP can be solved and thus a solution for the POMDP can be obtained.

2.3.2 RL

The section presents the concept of Reinforcement Learning

Definition

Reinforcement learning (RL) is a machine learning technique that is able to solve MDPs where you don't have access to a complete model of the environment, that is; T and R are not entirely known. The general idea is to let an agent act in and thus explore its environment, and generate a policy to follow based on its experience.

Learning

Here we will present the most common algorithm for solving RL problems, Q-learning. This is also the algorithm we will employ in our experiments.

Q learning can be described as value iteration through exploring. We define the function $Q(s, a)$ as the maximum expected, discounted cumulative reward the agent can receive from choosing action a in state s. From this, the optimal policy π^* can be obtained by choosing the action with the highest expected reward in each state:

$$\forall s\, \pi(s) = \arg\max_a Q(s, a) \tag{2.6}$$

The Q learning algorithm seeks to learn a function $\hat{Q}(a, s)$ which estimates $Q(a, s)$. This approximate Q function can then be used to generate the approximate optimal policy $\hat{\pi}^*$.

When learning the Q function the learning program keeps a table of temporary values for each $\hat{Q}(s, a)$. It then iterates over choosing a state s, executing an action a and observing the resulting state s' and reward r (if s' is absorbing, a random new state is chosen for the next iteration). For each iteration it updates the value for $\hat{Q}(s, a)$ with the immediate reward plus the current maximum value resulting performing the optimal action a' in state s' discounted by a factor γ. We use the notation $\hat{Q}_n(a, s)$ to denote the value of $\hat{Q}(a, s)$ at iteration n. The algorithm can be expressed as the rule in Equation 2.7. $\hat{Q}_n$ can be proven to converge towards the actual Q if each state-action pair is visited infinitely often, and it may require many thousands of iterations to accomplish a semblance of convergence in practise.

$$\hat{Q}_n(s,a) \leftarrow r(s,a) + \gamma \max_{a'} \hat{Q}_{n-1}(s',a') \tag{2.7}$$

The training rule presented in Equation 2.7 works for deterministic environments where performing action a in state s always results in the same state s' and the same reward r. This is not the case for the experiments we will be performing in Soccer Server. Due to the uncertainty introduced on purpose by Soccer Server and that fact that the other players on the team, and the opposing players are also performing actions, we can not be certain that performing a given action in a state will result in the same reward and state every time. This means we will have to use a training rule for our Q learner that is generalised to work in non-deterministic environments. The generalised training rule updates the $\hat{Q}$ values with a decaying weighted average of the current $\hat{Q}$ and the revised estimate as show in Equation 2.8.

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n(r(s,a) + \gamma \max_{a'} \hat{Q}_{n-1}(s',a')) \tag{2.8}$$

where

$$\alpha_n = \frac{1}{1 + \text{visits}_n(s,a)} \tag{2.9}$$

$\hat{Q}_n(s,a)$ is then the average of all the updates performed to $\hat{Q}(s,a)$ up to iteration n. Given that Equation 2.7 converges to Q, so does Equation 2.8. $\hat{Q}$ converges very slowly to the actual value of Q however. If the agent uses a thousand iterations to discover that an action eventually leads to victory, α will be 0.001 and so will require another ten thousand iterations of only that path to approach 90% of Q. Fortunately, absolute values of Q are less important than the relative differences, which $\hat{Q}$ will provide.

Action selection is crucial in Q learning in order to get a good balance between exploiting actions we have experienced give good results and exploring new actions we don't have as much experience about. A common action selection policy is k-selection where you use a variable $k \geq 1$ to choose actions according to the following equation:

$$P(s,a) = \frac{k^{Q(s,a)}}{\sum_{a'} k^{Q(s,a')}} \tag{2.10}$$

Choosing k to be 1 makes every action equally probable, and thus very exploring, and increasing k will make the policy more exploiting and less exploring. A recent method for choosing actions in reinforcement learning is UCT algorithm described in Section 2.3.3. Due to its very good results when employed in problems with large state spaces we will use this algorithm.

Q learning suits our purpose well, as the learning algorithm does not require any knowledge of which states or rewards an action will result in, knowledge that we do not have in our experiments. The MDP presumptions, however, are not as well covered. This is discussed in Section 2.3.5.

Constraints

The most important constraint of RL is that it is not feasible to solve the problem in practise if the state space and/or the action space grows to large. As described in the Section on solving RL problems the agent has to visit each state/action pair an infinite number of times to guarantee convergence, and this is hard to approximate with current technology if the state/action space is to big. Using UCT for action selection will help us with this problem as it guides our search through the state/actions space in a way that has proved to

2.3.3 UCT

This section presents Upper Confidence bounds applied to trees

Definition

UCT, short for Upper Confidence bounds (UCB) applied to Trees, is a recent method for planning in large state spaces (Kocsis & Szepesvári, 2006). It's a Monte-Carlo based planning algorithm which treats a state as a K-armed bandit problem, using the Upper Confidence Bounds (UCB) algorithm UCB1 for choosing actions. First we will describe the UCB1 algorithm and then define UCT.

UCB1 definition UCB1 is an algorithm for choosing which arm to play at a specified time on a K-armed bandit, also called an allocation policy. The K-armed bandit problem is defined by the sequence of random payoffs. If we let i denote the index of the arm played and t as the time step we pulled the arm, the payoff can be expressed as $X_{i,t}$. So if you pull arm number two the fifth time you play, you get the reward $X_{2,5}$. An allocation policy A is a policy defining which arm to pull at a given time. The performance of A is usually measured in its *regret*, that is, the difference between the summed expected payoffs when using the optimal policy and when using allocation policy A. For a large class of payoff distributions, there is no policy whose regret grows slower than $O(\ln n)$ (Lai & Robbins, 1985). The regret of UCB1 has a growth factor that is smaller than $O(k \ln n)$ (Auer & Cesa-Bianchi, 2002), and is thus said to solve the exploration-exploitation trade-off in K-armed bandit problems.

UCB1 keeps track of the average reward for each arm. If we define the index of the arm pulled at time t as I_t we can express the number of times arm i has been pulled up to (and including) time t as $T_i(t) = \sum_{s=1}^{t} \mathbb{I}(I_s = i)$. We then get this expression for the average reward we have obtained from playing arm i: $\bar{X}_{i,T_i(t-1)}$. We assume that X_{it} are independent for a fixed i and identically distributed and that they are almost surely bounded.

We want to create a one-sided confidence interval so we can choose the action with the highest upper confidence bound. To do this we need to construct a bias-term $c_{t,s}$. Since

we want the confidence interval for an arm to shrink when we pull the that arm, and grow when we pull some other arm, we choose the following inequalities to be satisfied:

$$P(\bar{X}_{is} \geq \mu_i + c_{t,s}) \leq t^{-4} \tag{2.11}$$

$$P(\bar{X}_{is} \leq \mu_i - c_{t,s}) \leq t^{-4} \tag{2.12}$$

Applying Hoeffding's inequality to these we get the following bias term:

$$c_{t,s} = \sqrt{\frac{2ln(t)}{s}} \tag{2.13}$$

When UCB1 is prompted to choose an action, it sums the average reward and bias term for each arm, and chooses the one with the best upper confidence bound, that is:

$$I_t = \arg\max_i \{\bar{X}_{i,T_i(t-1)} + c_{t-1,T_i(t-1)}\} \tag{2.14}$$

UCT definition UCT is a Monte-Carlo based roll-out algorithm that is especially good at planning in problems with a large state space. It is based on a generic Monte-Carlo roll-out planning algorithm, as shown in Table 2.1. The algorithm iteratively simulates an action in the current state and observes the reward and following state which becomes the new current state until a given state/depth is reached (the *search*-function). While doing this it updates the q-values for each state. It calls the search-function as many times as it has resources to do, and then picks the best action from the original state.

Table 2.1: Pseudo code for a generic Monte-Carlo roll-out planning algorithm

Step	Description
1	**function** MonteCarloPlanning(*state*):
2	**repeat**
3	search(*state, 0*)
4	**until** Timeout
5	**return** bestAction(*state,0*)
6	**function** search(*state, depth*):
7	**if** Terminal(*state*) then return 0
8	**if** Leaf(*state, d*) then return Evaluate(*state*)
9	*action* := selectAction(*state, depth*)
10	(*nextstate, reward*) := simulateAction(*state, action*)
11	q := *reward* + γ search(*nextstate, depth* + 1)
12	UpdateValue(*state, action, q, depth*)
13	**return** q

UCT uses UCB1 in step 9, where it treats the action selection problem as a separate multi-armed bandit problem. The actions are arms, and cumulative future rewards are payoffs. If we define $Q_t(s, a, d)$ as the value of taking action a in state s at depth d at time t, $N_{s,d}(t)$ as the number of times state s has been visited at depth d up until time t and $N_{s,a,d}(t)$ is the number of times action a has been taken in state s at depth d up to time t, the algorithm chooses an action according to the following rule:

$$\arg\max_a Q_t(s, a, d) + c_{N_{s,d}(t), N_{s,a,d}(t)} \tag{2.15}$$

For a more thorough theoretical analysis of UCT we refer the reader to (Kocsis & Szepesvári, 2006)

2.3.4 Soccer server as a POMDP

The entire state of the world is not observable to the players in soccer server. This is due to the inherent noise that is added to all observations, and to the fact that it is not possible to observe the whole field at once as each player has a limited field of view, and a limited maximum view distance. Details about the object you can see also deteriorate as the object gets further away. You can for example not see the jersey number or even the team colours of a player that is far away.

For our first simple experiments there will not be enough interesting objects (players and ball) for it to be a problem to keep them all in sight, and as the noise added to observations is rather small, we believe that if we neglect the partial observability, and use an MDP framework to model the environment, we will still obtain near-optimal results. Treating the environment as a POMDP would also cause practical problems due to the large amount of information that the computer would have to store and process for each action selection. Our later experiments will have to take into account the limited observability though, and we plan to make a model of the world that keeps track of where the other players are, where they were heading, and how long it is since we have seen them. By annotating the agents state description with this information we again believe that an MDP representation will be sufficiently accurate for the agents to learn their behaviour. The BrainStormers state framework that is presented in Section 2.4.1 is designed like this, and they have successfully used it as an MDP environment.

2.3.5 MDP presumptions in soccer server

As stated in Section 2.3.4, we will treat the soccer server environment as an MDP, and in an MDP there are three important presumptions that must be present in the domain that is being modelled. This section will present a short discussion of the Markov property presumption, the presumption that the policy π is a direct function of the states S and the presumption that the resulting state s_{t+1} given the current state s_t and action a_t is independent of t in our RoboCup. We will also discuss how we can handle the partial observability in our domain.

Markov property

The Markov property is that the resulting state s_{t+1} only depends on the current state s_t and action a_t. To make this hold in the soccer server setting the state space must be carefully constructed to encompass enough of the world state to render the past irrelevant. It is impossible in practise to make it hold completely, as the position of the other players, controlled by other agents, are so important to the game state, and the location of these players will depend upon the actions the agents controlling them chooses, and thus violating the Markov property for our agent. You can minimise this discrepancy by incorporating both the location, the velocity and the assumed current intention of the other players in the state description, but accurately predicting the intention of other players can be difficult in practise. The intention of a player running in the direction of the ball could be to run to the ball, or just run past it. A player running upfield with the ball could intend to get to the end line and pass the ball to a player in the penalty box, or it could be to dribble the ball to a good shooting position and shoot it towards goal.

Is the policy π a direct function of the states S?

This presumption says that for every state s there is a single action a that maximises the accumulated discounted future reward. As the soccer server setting is a multi-agent environment it is hard to prove that one action is definitely better to take in a given state than another. The optimal action in a given situation will depend on the actions taken by the other agents and the best result we can hope to achieve is to converge to the best Nash-equilibrium in the state/action space. Since all our agents will draw on the same knowledge base (state-action values table), and have trained together and thus learnt indirectly through experience what the other agents will do we believe that it is possible to find this equilibrium. Our belief in the viability of our method is strengthened by the fact that BrainStormers have used the same approach successfully.

Stationary

The presumption here is that the probability of ending up in state s when taking action a in state s' is the same no matter at which point in time the situation takes place. The problem it creates is that two situations that are otherwise inseparable in our state space should be handled differently due to the point in time they occur. The only way to overcome the problem is to define the states expressive enough to separate two situations that would violate the presumption if they were grouped as a single situation. It will also be helpful to save some data about the history of the game and use this in the state description.

Consequences of the MDP presumptions

To accommodate these three presumptions and at the same time make the experiments converge to an optimal strategy we have to find a middle ground when defining the state

space S. The states have to be expressive enough to define the states in such a way that the presumptions are not violated, but the state space must not become so large that training it to convergence becomes an unmanageable task. The BrainStormers framework have made a powerful state space definition that enhances the current state information with selected information based on the previous states. If using states based on the BrainStormers framework we will consider developing algorithms for predicting the intentions of the other players to further enhance our state-space.

Previous experience (Holen & Marøy, 2007) has shown that using UCT is a solution when working with state spaces that are unmanageable using traditional RL action selection schemes. This leads us to believe that if we are unable to converge to a solution when using UCT the size of the state space must be reduced. We could also have employed recovery strategies that can elicit working policies from unconverged learning, but that is beyond the scope of this project.

2.3.6 Learning in multi-agent environments

For a comprehensive survey of multi-agent systems see (Stone & Veloso, 1997). This section will present some of the issues of learning in multi-agent environments with homogeneous non-communicating agents, as this is the kind of agents we focus on in the project. This means that all agents have the same internal structure, i.e.; they are clones. They will act differently from each other because they are situated in different places, and thus experience different situations. The procedure for selecting actions from their common action-set is also identical for all agents.

We will look at four issues that arise in the multiple homogeneous non-communicating agents setting, and discuss how we will handle them in the soccer server setting. Reactive vs. deliberative agents, local vs global perspective, how agents can model other agent's states, and how agents can affect other agents.

Reactive vs deliberative agents

An agent can be reactive, and choose its actions directly based on the sensor input it obtains, like a human reflex, or it can be deliberative and keep an internal state and predict the actions of other agents, and their effects. These are two extremes, and agents can be constructed to lie anywhere between the two. Although we plan to keep an internal state for our agents in our experiments, they lie close to the reactive end of the scale, as they do not directly predict actions or the state of other agents. The agents will move further towards deliberative if we implement algorithms for predicting the intentions of other agents or if we implemented a longer look-ahead search in our UCT implementation.

Local or global perspective

Agents with local perspective only obtain input from its immediate surroundings, while one with global perspective will obtain information about the global state of the world.

Global perspective is better in terms of making the Markov property hold better for our agents, but global perspective often leads to problems when too many agents observe the same weakness and all try to correct it. Local perspective often works better in such situations as agents observe different local weaknesses they can help with. Our agents will use approximately global perspective since the location of the player is such an important part of the agent's state, and agents can not occupy the same location. We assume this will provide enough difference in the agents' views of the world to avoid them making the same decisions.

Modelling states of other agents

To work better together agents can model the state and/or sensor input of the other agents, and try to predict the actions they will take. This approach can also be done recursively (what does agent A think agent B thinks agent A will do). Instead of modelling the state of other agents we will design our agents to treat the other agents as part of the environment. Previous experiments have shown that this approach works when trying to make agents cooperate (Schmidhuber, 1996)

Affecting other agents

Agents can affect other agents in several ways. Actively they can be observed by the other agents and they can change the state of other agents. Inactively they can affect them by active or passive stigmergy. Active stigmergy occurs when the agents changes the environment with the intent of making other agents observe the change. This can for example be done by placing markers. Passive stigmergy is to change the environment in such a way that the actions of other agents get different outcomes. For example, if one agent removes a fuse, it will change the outcome of another agents action ”turn on light switch”. We do not plan to that our agents will actively try to affect other agents in any other way than being observed by them, but we might have to use the speaking-capabilities in soccer server to facilitate better cooperation between our agents.

2.4 The state of the art

This section describes the current state of the art of the RoboCup 2D soccer simulation league. It is focused on BrainStormers, a RoboCup team currently from the University of Osnabrueck in Germany, by Martin Riedmiller, Thomas Gabel and others. BrainStormers have a very high standing, coming in the top three since 2000, and winning in 2005 and 2007. It's highly relevant, since it uses reinforcement learning for strategy, and the authors willingly reveal their techniques in publication, even to the point of releasing most of the source code of their 2005 version. It therefore makes an ideal examples for study.

Figure 2.4: BrainStormers logo

2.4.1 BrainStormers

BrainStormers consists of two main parts, described in (Riedmiller & Gabel, 2007): the world module and the decision module. The world module reconstructs the approximate world state based based on current and past sensory input, which the decision part uses when determining moves.

The decision part consists of a large number of components, each responsible for performing some action. An action can be low level or high level, and invoke other actions. Examples are modules for searching the ball, running to specific areas of the field, dribbling and kicking. Some of the modules are hand written, while others are learnt, depending on which gives the best end result. Lower level skills, such as intercepting the ball, are learnt separately, and used as an action when learning high level strategies such as scoring goals.

Learning low-level skills

When BrainStormers learn low level skills, different scenarios are generated for the agent to learn from. The action set depend on the skill being learnt; for intercepting the ball, it might consist of moving in one of eight directions. The state consists of the position, direction and speed of the ball, and the position of teammates and opponents, as tracked by the world module. Rewards are given when the goal is achieved, such as the ball leaving the player at certain angle and speed.

Learning high-level behaviour

In learning high level behaviour such as ”with ball behaviour”, or ”no ball behaviour”, previously learnt low level skills such as passing to a teammate or dribbling are used as actions. The result of the high-level learning is then an appropriate sequence of low-level actions to perform in order to accomplish the given task.

Detailed description of BrainStormers action selection

The framework for deciding what to do is made up of modules called Behaviours. These are organised in a non-strict hierarchical structure from high level behaviours like ”defencive behaviour” and ”aggressive behaviour”, to low level behaviours like ”goalshot

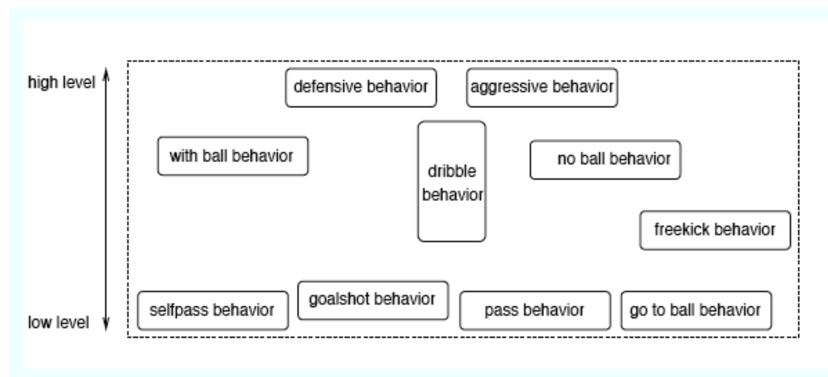

Figure 2.5: BrainStormers behaviour hierarchy

behaviour" and "pass behaviour". This organisation is presented visually in Figure 2.5. Behaviours can choose which action or actions that are to be performed themselves or delegate all or some of the action choosing to other behaviours. The organisation is non-strict as delegation can go both ways. A high level can delegate to a low level, or a low level can delegate to a high level. An example of the latter is that the skill of intercepting the ball can decide that intercepting is not a good idea, and delegate the responsibility to the defencive behaviour instead (Riedmiller & Gabel, 2007). The actual delegation is implemented by each behaviour having a function get_cmd(Cmd& cmd) that can be called by other behaviours, and passed the reference to the command cmd. What a certain behaviour chooses to do depends on the state of the world at the time of choosing. Each behaviour is implemented as a separate class in the source code.

The framework keeps track of the current state of the world in the class ws and also keeps information about previous world states in the class ws_memory. ws contains information about each player and the ball, such as location, velocity, body angle and neck angle. It also keeps track of how long has been is since the observed player/ball was last observed by the observing player. It also has extra information about the observing player, such as view angle and quality, and general information about the game such as play_mode and current score. ws_memory keeps information that is not observable in the snapshot contained in ws such as which team is attacking, which player last had the ball and information about the opponents offside line. There is a separate class ws_info that is used to extract and compile information from ws, and it is ws_info and ws_memory that are the sources of information that the various behaviours use when deciding which action to choose, or to delegate the choice to another behaviour. Using the information from ws_info and ws_memory makes the setting behave more like and MDP than without it. Even though information about the environment might be out of date and not correct at all times, you have a complete set of inaccurate information to base your decisions on rather than an incomplete set of less inaccurate information. no information will be completely accurate due to the inherent noise in soccer server.

The behaviours will ultimately call on one or more basic skills such as "go to position"

(neuro_go2pos) or "kick ball" (neuro_kick_05). It is important to know that the initial get_cmd(Cmd& cmd) can lead to many low level skills being performed in an optimised order, and not just to one single skill being performed. These are the low level skills in the BrainStormers hierarchy that are learnt as described in section 2.4.1

As soccer server allows you to perform three actions in each cycle (one basic-, one neck- and one view-action. BrainStormers has a set of behaviours for each category. The three sets are loosely coupled as some of the low level basic skills requires specific neck or view-behaviours to be available in order to function optimally.

2.4.2 Multi-agent reinforcement learning

This section will present various techniques for doing multi-agent reinforcement learning.

Modular Q learning

(Park, Kim, & Kim, 2001) has used modular Q learning for learning team play in robot soccer (NaroSot). It uses a mediator module that uses the Q values of the individual learning modules as input, and chooses an action to deal with the state explosion in multi agent learning tasks. Mediation deals with only two actions: to remain in position, and to go kick the ball. Because of the use of a mediator module, this technique isn't directly implementable in RoboCup, where the agents can't communicate outside of the simulation.

Agents have their individual home positions, in a 1-2-2 setup. Mediation is used when the ball is between two home positions, to decide on which agent should chase the ball. This arbitration is based on the agent's learnt confidence in the action, its distance to the ball and the difference in angle. More specifically, the action is chosen by

$$\text{argmax}_a \; f(Q_i(s_i, a_i), \theta_i, d_i) \tag{2.16}$$

where i is the agent's number, $Q(s,a)$ is the Q function for the state s and action a as described in (Watkins, 1989), θ_i is the angle between the agent's direction and the direction to the ball, and d_i is the distance to the ball. In the experiments in NaroSot, f is simply the sum of the weighted arguments, $f(q,\theta,d) = 5/10q + 3/10\theta + 2/10d$.

The mediation ensures that only one agent – the one who can most quickly reach the ball – selects the kick action. The other will be forced to remain in its position, regardless of what its Q function says.

Other irrelevant multi-agent Q learning techniques There are many more examples of Q learning employed in multi-agent environments where the constraints for decision making or available information about the environment does not correspond to what we are dealing with in this project. One of the most notable examples is the work of Carlos E. Guestrin on coordinated reinforcement learning (Guestrin, 2003). He presents 3 multi-agent reinforcement learning algorithms that perform well in empirical

evaluation, but the algorithms make use of extensive inter-agent communication in making combined Q-functions for several agents and in action selection. The results are thus not applicable for our project.

2.4.3 State of the Art in contemporary soccer games

In non-RoboCup soccer games, such as Electronic Arts' FIFA series, the AI is fairly simple. According to Matt Torvalds, a former EA AI developer, players are controlled by simple decision trees. The leaves of the trees are sets of possible actions, where one action is selected at random. A number of different trees are available, representing different personalities such as ”aggressive” or ”defencive”. Some learning is applied to the decision trees, which are saved between games. The goalie would use a more sophisticated algorithm like minimax.

These techniques are designed to make the players mimic human behaviour as much as possible, and use the available perfect information and player/ball control to make the game appear real rather than to simulate it realistically. The methods are therefore not very relevant for RoboCup.

Chapter 3

Architecture

This Chapter will present the architectures of our implemented agents. It will present it in Section 3.1

3.1 Colugo

This section describes the architecture of the Colugo agent.

3.1.1 Overview

Figure 3.1 shows the main components in the Colugo experiments, from Experiment 3 and up.

RoboCup tournaments and experiments are based on an official soccer server, as described in Section 2.2.1. BrainStormers connects to the soccer server, to send commands and receive input for each simulated player. The high level decision making is delegated to a UCT learning component. The UCT learning component also include a

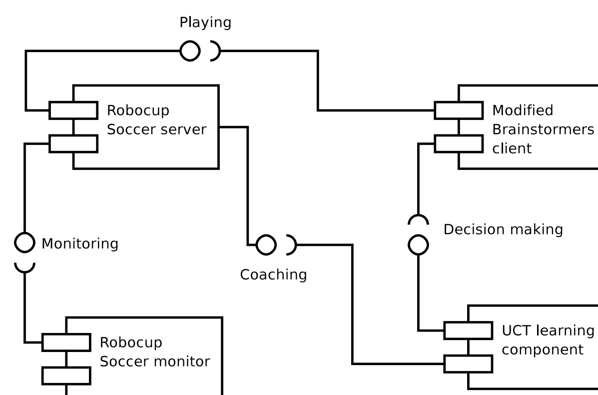

Figure 3.1: Main components in the Colugo experiments

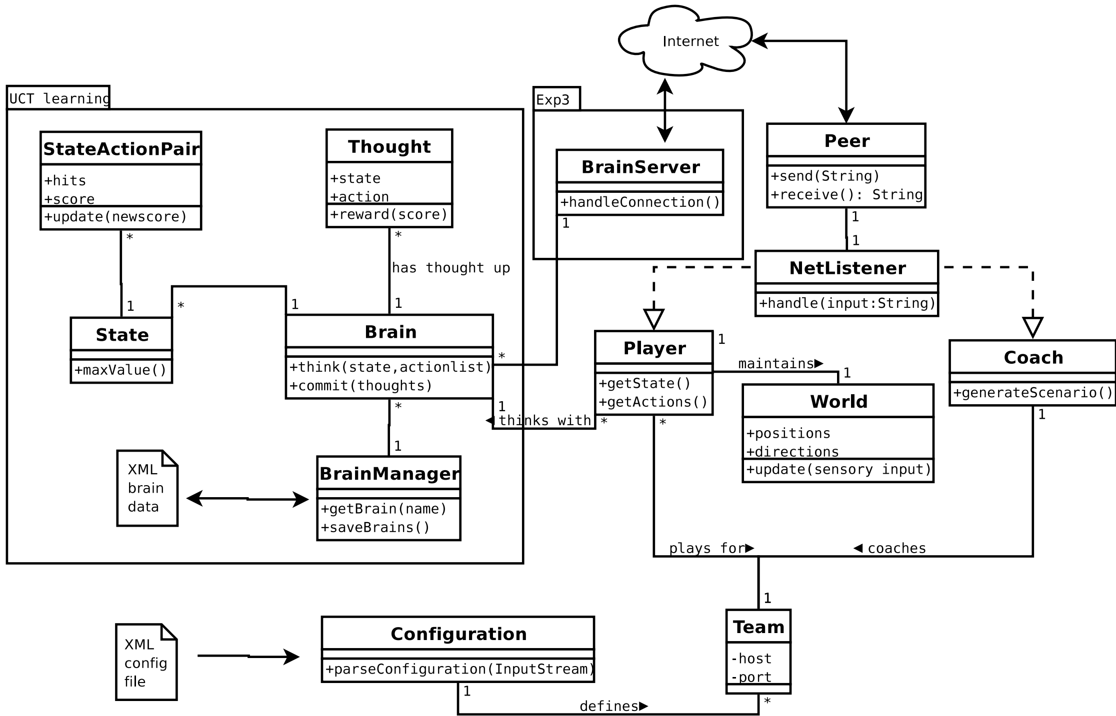

Figure 3.2: UML diagram of Colugo's learning component

coach that generates different starting scenarios by placing the players and the ball at random position, and monitor their progress.

In Experiments 1 and 2, the UCT learning component also handled player input and output, as described in Section 3.1.2.

3.1.2 The UCT learning component

Figure 3.2 shows a simplified UML diagram of Colugo's UCT learning component, and the player agent from Experiments 1 and 2. The component is implemented in Java.

The first experiments were defined by an XML configuration file. The file specified some teams, each with a coach and some players. Players were given a Brain, which would handle UCT learning. Multiple players could share a Brain between them, or they could have their own. Players received input and sent commands directly to the soccer server. A World class was used to keep track of previous information and triangulate positions in Experiment 2.

After the experiments with low level actions failed, we added a BrainServer to allow remote access to the UCT learning from a modified BrainStormers client, to avoid having to reimplement the system in the BrainStormers C++ framework.

The Brain class uses UCT to select an action based on a state description (an arbitrary string), and keeps the learnt data in the form of State objects full of State-Action

pairs. Learning is not applied until a list of decisions (known as a train of Thoughts) have been committed. This allows results to be propagated backwards immediately to speed up learning, as described in (Holen & Marøy, 2007).

3.1.3 The modified BrainStormers client

As described in Section 2.4.1, each BrainStormers behaviour has a function get_cmd to return commands. We created our own behaviour which asks the BrainServer for decisions over a TCP socket. The behaviour gathers a state description (described in each experiment) from the BrainStormers world view, along with a list of allowed actions. The BrainServer feeds this to the Brain, which returns a result. The BrainStormers client also sends reward/commit messages when goals are scored.

When an action is chosen, such as intercepting the ball, kicking it, or running to a location, the relevant behaviour module is given control until it considers itself finished, or until a timeout is reached. A new action is then requested.

This new behaviour, aptly named the Colugo behaviour, replaced the entire BrainStormers behaviour hierarchy in our modified agent. A significant difference between our behaviour and the old hierarchy is that in our agent each call to the behaviour results in one low level skill being performed, while in the original agent one call to the behaviour at the top of the hierarchy might lead to several low level skills being performed in an optimised order.

Chapter 4

Experiments

4.1 Introduction

This section will document the main part of our project, the experiments we will conduct in an effort to teach our agents to play robot soccer in the RoboCup setting using reinforcement learning with UCT. We plan to start out with as few restrictive measures as possible and hope to see and emerging intelligence. If this turns out to be unfeasible we plan to gradually impose restrictions and higher level actions until we get a setting in which it is practically possible to conduct learning.

We plan to gradually increase the complexity of our experiments starting with very basic ones to test our settings and expanding with more teammates and opponents. We expect to get good results in a 2 vs 2 setting by the end of our project, and hope to get a chance to try 3 vs 3.

4.1.1 Experiment infrastructure

The experiments will be run on a desktop machine with a Intel Core 2 Quad processor, 4 GB RAM and running the Linux distribution Debian Etch 64-bit as operating system.

The rest of this chapter consists of section 4.2 through 4.15 presenting experiments 1 through 7 and their results.

Players:	1
Opponents:	0
Key feature:	States defined directly from input

Table 4.1: Summary Experiment 1

(dash 100)	Run forwards
(dash -100)	Run backwards
(turn 30)	Turn 30 degrees left
(turn -30)	Turn 30 degrees right
(turn 60)	Turn 60 degrees left
(turn -60)	Turn 60 degrees right
(kick 0 100)	Kick the ball straight ahead with full power

Table 4.2: Actions in Experiment 1

4.2 Experiment 1 – Translating vision directly into states

4.2.1 Setting

We will start with a single player and the ball randomly placed in front of the opponent's goal. The player will get 20 seconds to score. The experiment is summarised in table 4.1.

4.2.2 States

The states are simply the discretised versions of the vision input. For a see-command like "(see 9 ((f c t) 75.2 -70) ((f r t) 65.4 -27) ((f r b) 14.2 68 0 -0) ......)", we sort by angle and round distances to the nearest 5 and angles to the nearest 15, giving a string like "fct@-75:75,frt@-30,65,frb@75:15".

4.2.3 Actions

We start with the basic actions described in Table 4.2, which are direct inputs to the soccerserver.

4.2.4 Weaknesses

This is a very directly defined state space, and it might be too big since there are on average 20 flags visible at any time.

4.2.5 Expected results

We hope the agent will be able to score at least the easiest cases when it just has to run and kick, and not deal with positioning itself.

Players:	1
Opponents:	0
Key feature:	Position is based on triangulation

Table 4.3: Summary Experiment 2

4.3 Experiment 1 - Results

4.3.1 Training results

We ran the experiment for 24 hours, with ten simultaneous instances. The agent only scored 6 times out of about 40 000 chances. There were about 400 000 visited states where $\frac{3}{4}$ of them were only visited once. This clearly indicates that the state space is unmanageably large.

4.3.2 Problems with the experiment

The experiment used extreme amounts of memory, due to the size of the state space. Theoretically you should see the same flags when you stand at a certain point and look in a certain direction, so even if the state space contains a lot of information, they are all functionally dependent on these two parameters. However, the deliberate fuzzying of input data from the soccer server ruins this and makes the state space exponentially larger.

4.3.3 Conclusion

Since the state space is too large, we have to find a way to reduce the number of states to learn. We could try learning a generalised mapping of $s \rightarrow Q(s, a)$, since in our simulation, $s_1 \approx s_2 \Rightarrow Q(s_1, a) \approx Q(s_2, a)$, but since there are an unnecessarily large number of dimensions whose only purpose is to provide means of positioning, we will instead extract this information analytically to simplify the learning and reduce the effect of fuzzy variables.

4.4 Experiment 2 – Triangulated position

4.4.1 Setting

As in Experiment 2, we have one player and the ball placed randomly around the opponent's goal. The experiment is summarised in Table 4.3.

4.4.2 States

Using all the seen flags, we triangulate the estimated position of the player and pick the average coordinates and angle. We also include the angle and direction to the ball as

(dash 100)	Run forwards
(dash -100)	Run backwards
(turn 30)	Turn 30 degrees left
(turn -30)	Turn 30 degrees right
(turn 60)	Turn 60 degrees left
(turn -60)	Turn 60 degrees right
(kick 0 100)	When near the ball, kick straight ahead with full power

Table 4.4: Actions in Experiment 2

the player sees it. The coordinates are rounded to the nearest half, and angles to the nearest 15 deg. This rounding is a simple way of interpolating Q-values of nearby points in continuous space, namely to assume that they are equal.

The state will be on the form "Me:X,Y@A Ball: b@B+D" where X and Y are the player's coordinates, A is the direction the player is facing, and B and D are the angle and distance to the ball.

4.4.3 Actions

The actions are the same as in Experiment 1, except now the player can only kick the ball if he's close to it. The actions are listed in Table 4.4.

4.4.4 Weaknesses

This appears to be the most direct and simplest way of specifying a state space without high-level actions, and still maintaining the information necessary for an MDP, but it's likely still too large. A rough estimate, where the agent occupies a small 30×20 area of the field , the estimated number of states is $2 * 30 * 2 * 20 * 24 * 25 * 10 \approx 15\ 000\ 000$, far more than what can be thoroughly learnt with current hardware.

4.4.5 Expected results

We hope the state space will show some improvement over the old state space, even if it's too large in itself.

4.5 Experiment 2 - Results

4.5.1 Training results

We ran the experiment for 24 hours, with ten simultaneous instances. The agent managed to score only four times, out of the 40 000 chances. The agent ran out of memory after approximately 400 000 visited states, most visited only once.

Players:	1
Opponents:	0
Key feature:	Integrating UCT into BrainStormers agent

Table 4.5: Summary Experiment 3

4.5.2 Problems with the experiment

The state space is simply too large. It's not feasible to learn soccer related skills in this setting.

4.5.3 Conclusion

Moving and turning in a continuum are not tasks suited for a low-level reinforcement learning agent. Our low-level states do not consider the laws of physics and geometry; using coordinates as states generalises movement from simple translation in a Cartesian plane to following edges in an arbitrary graph where each node is a point in space.

One possible route from here is to divide the field into larger areas and use another learning technique such as a neural network or case-based reasoning to estimate Q-values within the region. However, since this moves the focus away from UCT, it falls outside this project, so we will leave it as further work. Instead we will give up on low level actions, and use higher level ones instead.

4.6 Experiment 3 – Using the BrainStormers framework

4.6.1 Setting

This experiment will utilise the framework made by BrainStormers, see Section 2.4.1 for details on this. To interoperate with the framework we changed the main behaviour class to connect to our java agent Colugo, and let it perform action selection. Based on the response the main behaviour called the low level skill that Colugo decided upon. The resulting score was fed back to the java agent who updated the corresponding state/action score and thus performed the actual learning. We only used Colugo to select basic skills and let BrainStormers deal with the neck and view controllers. For more technical details about the integration see Section 3

As with the previous two experiments the goal will be to learn to score with one player and no opposition. The experiment is summarised in Table 4.5.

4.6.2 States

The state space will be based on the information gathered from BrainStormers world model, or world state. We will divide the playing field into 9 areas as shown in Figure 4.1. The areas are denoted by XY where X is the zone (Defencive, Middle, Attacking)

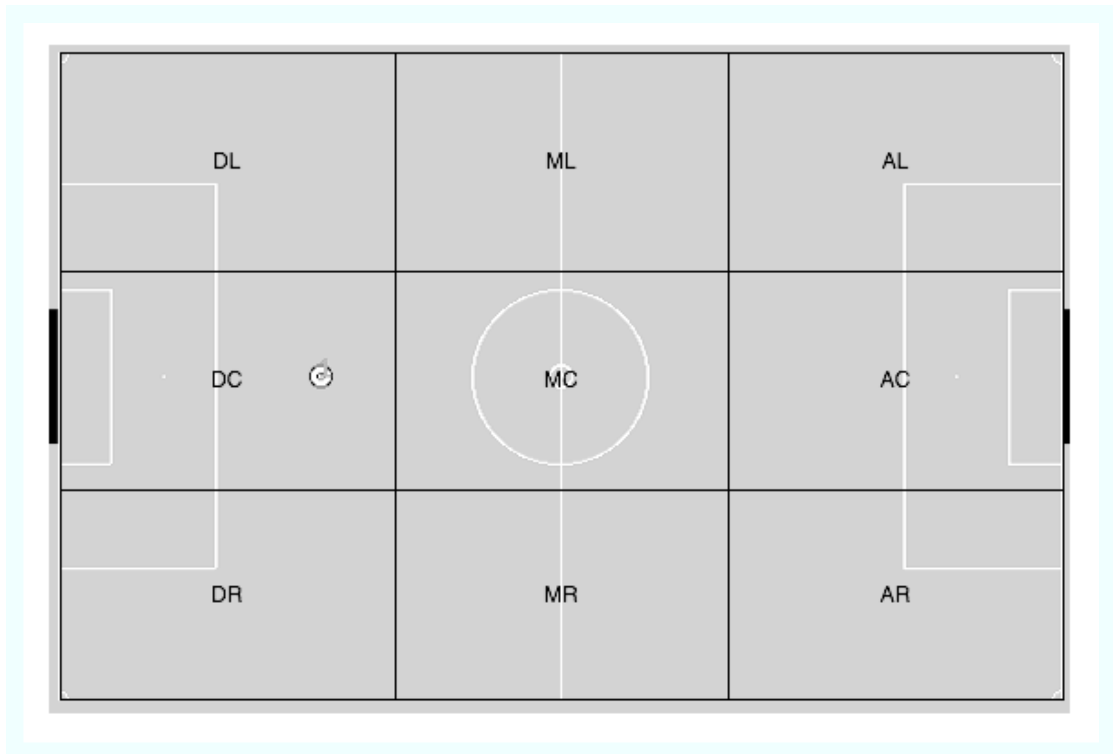

Figure 4.1: Field division in Experiment 3

and Y is the side of the field (Left, Centre, Right). The state will also include the area the ball is located in, if known, and if the ball is currently kickable by the player.

4.6.3 Actions

The action set will not be large for this experiment. It will consist of locating the ball, intercepting the ball, dribbling towards goal, and kicking the ball towards goal. Each action corresponds to invoking a BrainStormers skill for some number of cycles, and are listed in Table 4.6.

FACEBALL	Look around to locate the ball
INTERCEPTBALL	Run towards the ball, taking its speed and heading into account
DRIBBLETOGOAL	Dribble the ball towards the opponent's goal
KICKBALL	Kick the ball in the direction of the opponent's goal

Table 4.6: Actions in Experiment 3

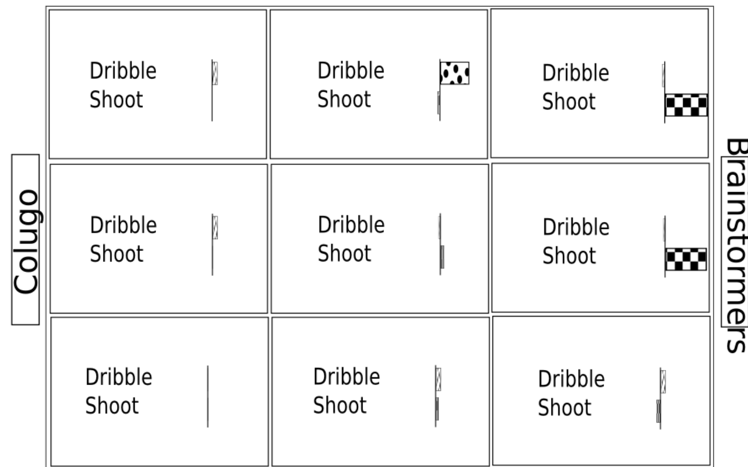

Figure 4.2: Ball behaviour per field division

4.6.4 Weaknesses

This is a very simple experiment, and we can not foresee any weaknesses.

4.6.5 Expected results

We expect the agent to learn to score goals in an efficient manner, as the actions are specifically tailored towards this, and there is no opposition.

4.7 Experiment 3 - Results

4.7.1 Training results

The experiment ran for approximately 24 hours, and the agent learnt when to dribble towards goal, and when to shoot in a satisfactory manner. If the agent didn't know where the ball was, its actions were restricted to finding the ball. If it knew the location of the ball, its only option was to run to it. As there were no opponents or players on its team, no other actions were meaningful to contemplate. This meant that the only situations with any real choice was the ones where the agent was able to kick the ball.

4.7.2 Problems with the experiment

The main problems with the experiment concerned the low level skills we used from the BS framework. In particular the Neurokick05 skill we used for kicking the ball towards goal, and the dribblebetween skill we used to dribble towards goal.

The kicking skill didn't consistently kick the ball in the direction we instructed it to kick in. This lead to some interesting results in the learnt policies; The two mirrored

Players:	2
Opponents:	1
Key feature:	Playing against opposition

Table 4.7: Summary Experiment 4

states where the agent had the ball in either the AL or the AR positions should intuitively produce the same results for the two actions dribble towards goal and shoot towards goal, but our agent found that when in AL, the best action to take was to shoot, while in AR, the best action was to dribble towards goal. The Neurokick05 skill uses an internal neural network that we have no easy way of inspecting, and we suspect the inconsistencies we observed can stem from asymmetry in the network.

When observing the agent we saw that when employing the dribble skill it often ended up standing on the same spot, kicking the ball around inside its range. This behaviour somewhat crippled the usability of the skill, but it did not always occur. We can not find the reason for the behaviour, but the skill is designed for dribbling in an environment with opponents, and the lack of these in our setting might throw the skill off.

Both these effects are visible in Figure 4.2, which lists the average score per action for a player with the ball in a given part of the field.

4.7.3 Conclusion

The experiment was a success, as the agent managed to score on a regular basis, and learnt to dribble towards goal when far from it, and shoot towards goal when close. The most important result though was that it showed that our integration with the BS framework was successful.

4.8 Experiment 4 – 2 Colugos VS 1 BrainStormer

4.8.1 Setting

After the successful integration with the BrainStormers framework we experience in Experiment 3, we want to increase the complexity of our environment and see if our agent can learn how to play together with a copy of itself. We will pit two of our agents against one BrainStormers agent. The BS agent uses a demonstration behaviour that are described as simple but effective by its makers, and will serve us well as opposition. It is a field player, not a keeper.

Our agents will get a big reward for scoring a goal, a similarly big negative reward for conceding a goal, and a small negative reward if the ball goes out of bounds. The experiment is summarised in Table 4.7.

FACEBALL	If the ball position is unknown or outdated, try to locate it.
GOTO *X Y*	Go to the specified position. One such GOTO action is defined for the centre of each of the field areas.
INTERCEPTBALL	If not near the ball, run towards it.
DRIBBLETOGOAL	If near the ball, dribble it towards the opponent's goal.
KICKBALL	If near the ball, kick it in the direction of the opponent's goal.
PASSBALL	If near the ball, kick it towards the nearest teammate.

Table 4.8: Actions in Experiment 4

4.8.2 States

We will have to expand on the state description and on the available actions to accommodate this new setting. We will keep the division of the game field into nine areas in the same way as in Experiment 3. The state description will contain information about the following variables: the agent's location, The known number of teammates (excluding the agent in question) and the number of opponents in each of the areas, the location of the ball, if the agent is the closest teammate to the ball, if the ball is currently kickable by the agent, and if the agent's team is currently attacking or defending.

4.8.3 Actions

The action set from Experiment 3 will be expanded with actions for going to the centre of each area and an action to pass the ball to the agent's teammate. As with Experiment 3 the actions that include kicking the ball (shooting, dribbling, passing) will only be available to the agent if the ball is kickable. Similarly, the action to locate and face the ball will only be available if the location of the ball is unknown or invalid. The actions are listed in Table 4.8.

4.8.4 Weaknesses

The most important weakness in this experiment is that the crude division of the field into 9 areas might not be descriptive enough to separate similar situations that should be handled differently. The agent also has no way to know when the other agent will catch a pass, or when it will perform a pass. This might lead to poor cooperation and the agent to learn that passing the ball is a always a bad idea.

4.8.5 Expected results

We expect the agent show signs of cooperation, and to manage to score in a majority of its tries against the BrainStormers agent.

4.9 Experiment 4 - Results

4.9.1 Training results

The experiment ran for approximately 60 hours with 15 soccer servers with 2 Colugo agents each, resulting in a total training time of 1800 hours. Our agents performed well when trained, managing to score in approximately half of the games against one BS agent. In 40% of the games the BS agent scored, and the rest of the games ended in the ball going out of bounds. Most of Colugo's goals came after a duel for the ball between one or both our agents and the BS agent where the ball ended up going towards BSs goal, when Colugo managed to pick up the ball afterwards, it dribbled towards goal and scored. When both Colugo agents were involved in the duel, they had a higher chance of winning it, so being two against one turned out to be a big advantage in this crucial game-situation.

4.9.2 Problems with the experiment

Observing the trained agents we could see very few signs of cooperation between the Colugo agents. When an agent passed the ball towards the other agent, the receiver was generally occupied with performing an action, and unless this action happened to be to intercept the ball, the pass was not received. Despite the fact that this rather unlikely combination was needed to get any cooperation, we saw several examples of beautiful passes resulting in goals.

The problems we had in Experiment 3 with the BS framework low level skills also occurred in this experiment. Kicks towards goal sometimes went awry, and dribbling towards goal often ended up in stationary agent behaviour. The dribbling problems was not as crucial this time, as the BS agent always came to try and take the ball, and this ended the stationary behaviour, either with the ball being kicked in a random direction, or one of the agents dribbling the ball away from the other.

4.9.3 Conclusion

Our agents managed to learn a behaviour that performed well against one opponent, but they did not learn to cooperate much. We must change our environment to better facilitate cooperation so our agents might learn that passing the ball might be better than just dribbling towards goal with an opponent on its heels.

Our 9 area division of the field worked satisfactory, but we did not try any other division schemes to compare. A more fine grained division might lead to more sophisticated behaviours, but as our agents did not learn to cooperate, and the best strategy seemed to be to run to the ball, and then dribble towards goal, we did not want to spend time on trying other field divisions in the setting of Experiment 3. If our agents manage to learn some cooperation in future experiments, it might be interesting to try other division schemes then, and do a comparison.

Players:	2
Opponents:	1
Key feature:	Passes interrupts the other player

Table 4.9: Summary Experiment 5

4.10 Experiment 5 – 2 VS 1: take 2

4.10.1 Setting

This experiment will be very similar to Experiment 4. It will only add functionality for interrupting an agents current action when another agent tries to pass the ball to it. To implement this in our agent, it will shout out who it passes to when it passes the ball. It will also continually listen for such shouts, and interrupt its action if someone calls out its name.

The experiment is summarised in Table 4.9.

4.10.2 States

The states will be enhanced with one more variable: Interrupted. This will indicate whether the agent was just interrupted by a shout. The variable will only be present in the state description if the agent was interrupted, and we can thus reuse the states and learning we did in Experiment 4.

4.10.3 Actions

No new actions will be introduced, so the list of actions is the same as the one described in Section 4.8.3. The only difference is that the passing action will be enhanced with the aforementioned shout.

4.10.4 Weaknesses

The field is still only divided into 9 areas, as described in Experiment 4.

4.10.5 Expected results

We expect our agent to learn that passing is a better option now, than in Experiment 4, and that it learns that it is smart to intercept the ball if the agent has been interrupted. The fact that we reuse the learning we did in Experiment 4 will make it easy to see if passing is a more attractive action now, as we can perform a direct comparison between two passing actions.

Players:	2
Opponents:	1
Key feature:	Agent interrupted by pass is forced to intercept ball

Table 4.10: Summary Experiment 6

4.11 Experiment 5 - Results

4.11.1 Training results

The agent trained for an additional 60 hours with 15 instances with 2 players each resulting in a total training time of 1800 hours. After learning it performed poorly in a test of 50 games with a win/tie/lose distribution of 20%/16%/64%. It did not learn any observable cooperation, and preferred the dribbletogoal strategy if it got possession of the ball. The new passball action had almost exactly the same Q value as the old one, -9.66 vs -9.56 on average.

4.11.2 Problems with the experiment

The agent failed to learn passing behaviour. This is because when one player passes, the other has to intercept for the pass to be effective. If the other player does not intercept the ball, which is the most likely case given the number of options it will randomly choose between, the first player will learn that passing the ball is a bad idea and will therefore probably not repeat it.

Since the receiving agent keeps learning and exploring, the result of passing changes over time for identical states. Q-learning will therefore not be able converge on an optimal strategy, and this might be what we encountered.

4.11.3 Conclusion

We should try a similar experiment, but force the other player to intercept incoming passes. When passes are more likely to work, we hope that some cooperation will emerge.

4.12 Experiment 6 – 2 VS 1: Forced pass-interception

4.12.1 Setting

Experiment 6 will add a bit of hardcoded behaviour to the setting used in Experiment 5 to get better cooperation between our agents. When an agent hears that it is being passed to, it will interrupt its current action and intercept the ball.

The experiment is summarised in Table 4.10.

4.12.2 States

The states will be the same as in Experiment 5. That is player location, ball location, number of teammates/opponents in each area, if the team is defending or attacking, if the ball is kickable, if the player is the closest teammate to the ball, and if the player has been interrupted.

4.12.3 Actions

No new actions will be added, the list of actions will be the same as in Section 4.8.3.

4.12.4 Weaknesses

The field is still only divided into 9 areas.

4.12.5 Expected results

We expect the agents to learn to cooperate and perform better than in Experiment 5. If we see that the hardcoded pass reception is working well, we might end the experiment before our agents are fully trained, and move on to a more complex setting.

4.13 Experiment 6 - Results

4.13.1 Training results

The experiment ran for approximately 64 hours with 15 instances with 2 agents each, resulting in a total training time of 1920 hours. The agents did not get fully trained in all states, and did not even encounter all states, but trained sufficiently in the important states where it was close to the ball, and we got enough results to end the experiment and move on. The agents learnt a good passing behaviour, especially around the middle of the field. The passing usually resulted in our agents having more stamina than the opponent, due to it running between our agents. Our agent was then able to dribble the ball ahead of the opposition towards goal.

In a small test of 50 games Colugo scored in 32%, BrainStormers scored in 30% and 38% ended in either the ball going out of bounds or the 60 second time limit expiring. These statistics is not reflecting the true situation, as our agents often had problems with scoring on an open goal, due to the previously mentioned imperfect performance of the actions dribble towards goal and kick towards goal. Many of the games ending in a tie would have ended in goals for Colugo if these problems were solved.

By the end of this experiment we had developed logging functionality for our coach, and could do larger test than the manual 50-game test. We let the agents play a game of first to 100 goals. BrainStormers won 100-88 (53%-47%) and the game lasted just over 194 minutes. This means approximately one goal per minute.

Figure 4.3 illustrates the average score for ball behaviour for an agent in a given region. The agent has clearly learnt that being near the ball by his own goal is far less

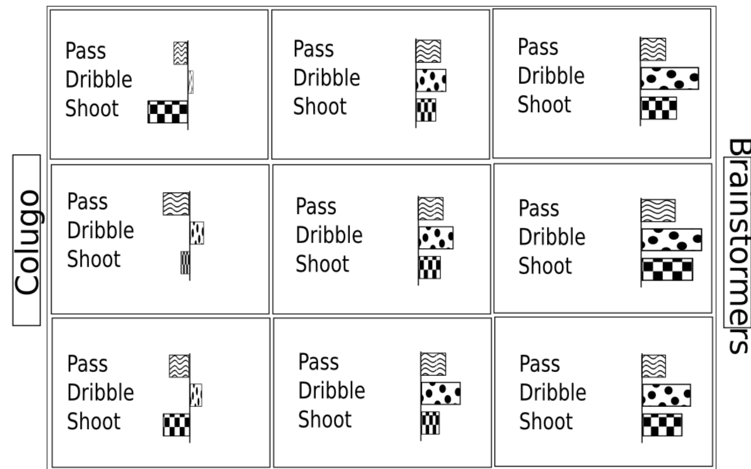

Figure 4.3: Ball behaviour per field division

favourable than being near the opponent's goal. We can also see that the agent has found out that passing is not a good idea near the goal, and that shooting is not favourable far away from the goal. Interestingly, however, shooting is not the best idea near the goal either, and the agent prefers to dribble to ball right into the goal. This is at least in part because of the uncertainty in hard kicks, which sometimes go in entirely unexpected directions. We can also see that as expected, there is lateral symmetry with only minor differences between the left and right sides of the field. Colugo approximately tied, and there was clearly significant learning.

4.13.2 Problems with the experiment

We still experienced the same problems as in previous experiments with the dribbling and kicking routines.

The combination of passes being passed towards the location of the receiver, and the receiver intercepting the ball by running towards it often lead to both our agents (and most often the opposing agent as well) ending up very close together after a few passes.

4.13.3 Conclusion

The experiment showed that our agents managed to outplay the opposition by passing and being forced to intercept passes. We are not interested in using a lot of time fully training our agents in a 2 VS 1 setting, and will continue with more complex settings now that we know that our passing scheme works. We should also find a solution to the huddling of agents resulting from passes and interceptions.

Players:	2
Opponents:	2
Key feature:	2 vs 2 with long passes

Table 4.11: Summary Experiment 7

FACEBALL	If the ball position is unknown or outdated, try to locate it.
GOTO *X Y*	Go to the specified position. One such GOTO action is defined for the centre of each of the field areas.
INTERCEPTBALL	If not near the ball, run towards it.
DRIBBLETOGOAL	If near the ball, dribble it towards the opponent's goal.
KICKBALL	If near the ball, kick it in the direction of the opponent's goal.
PASSBALL	If near the ball, kick it towards the nearest teammate.
LONGPASSBALL	If near the ball, kick it about 20m in front of the nearest teammate.

Table 4.12: Actions in Experiment 7

4.14 Experiment 7 – 2 VS 2

4.14.1 Setting

This experiment will increase the complexity of the environment by setting up two Colugo agents against two BrainStormers agents. After seeing positive results of emerging cooperation in the previous experiments we will now pit our agents in an even match against BrainStormers.

The experiment is summarised in Table 4.11.

4.14.2 States

The states will be the same as in Experiment 6 with player location, ball location, number of teammates/opponents in each area, if the team is defending or attacking, if the ball is kickable, if the player is the closest teammate to the ball, and if the player has been interrupted.

4.14.3 Actions

In an effort to combat the bunching of players occurring when passing in previous experiments we will add another action to the existing action set. It will be a long pass where the passing agent shoots the ball towards a point a set distance (20 meters) in front of the receiving agent. We envision this will help our agents to not end up in the same location after a couple of passes. The actions are listed in Table 4.12.

4.14.4 Weaknesses

The field is still divided into only 9 areas, and we expect that this might not be enough to efficiently counter the cooperation of the opposing agents. If our agent tries to cover an opposing agent by running to the same area as the opposing agent is in it will often end up a significant distance from the opponent, and not be able to intercept it. A more fine grained division of the field will reduce the possible distance between our agent and the opponent it is trying to cover. It will also make it more attractive to run to the area next to the one occupied by the opponent, for example to block its path towards goal, or to get a head start when intercepting a pass.

4.14.5 Expected results

We expect our agents to learn how to counter attacks from the opponent, how to outplay the opposition when in possession of the ball, and to win more games than the opponent in a post-learning test.

4.15 Experiment 7 - Results

4.15.1 Training results

The experiment ran for almost 250 hours, playing over 250 000 games and making 42 million choices. At the end, it scored about 16% of the goals, as seen in Figure 4.4. However, it also learnt to give BrainStormers increased competition as seen in Figure 4.5. At first, BrainStormers could score about 1400 goals per hour, but as Colugo learnt the game, this reached around 800.

Learnt behaviour with the ball is shown in Figure 4.6. This figure is skewed more towards the negative side than the equivalent figure in 4.3, since we now play against two opponents instead of one. It's interesting to see that passing long is the preferred action right in front of the goal. This is because geometrically, unless the other player is in defence, a long pass takes the ball into the goal. However, if the kick fails and goes off on a tangent, it's still a pass, so the teammate is certain to run after the ball. This is a very clever and unanticipated use of the action set.

As in Experiment 6 we let the agents play a game of first to 100 goals. This time BrainStormers won 100-19 (84%-16%) in just less than 119 minutes. This works out to approximately one goal per minute, the same as in Experiment 6.

4.15.2 Problems with the experiment

Due to a bug introduced when performing testing runs in the previous experiment this experiment ran with a UCT confidence interval of 0 for the first 60 hours, and thus behaved very greedily. The bug was fixed and we continued with the data collected, which was completely correct but now very narrow. Figure 4.4 clearly shows the effect. The fraction of goals by Colugo nearly doubled with the more explorative strategy.

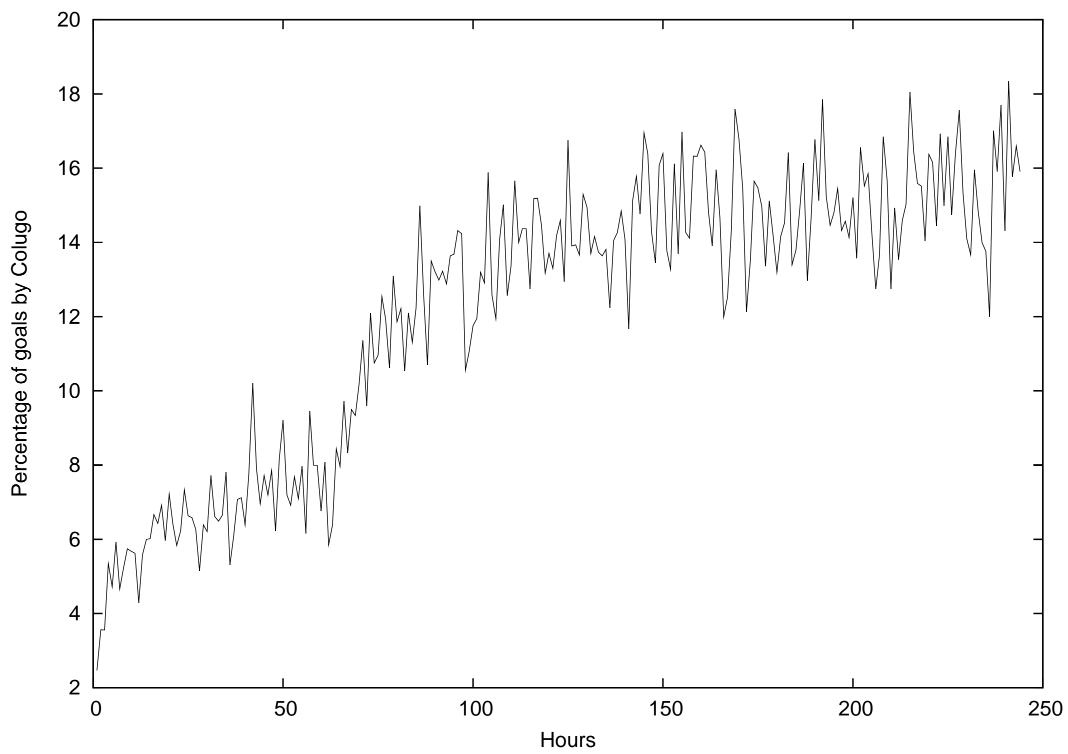

Figure 4.4: Exp. 7: The percentage of goals scored by Colugo

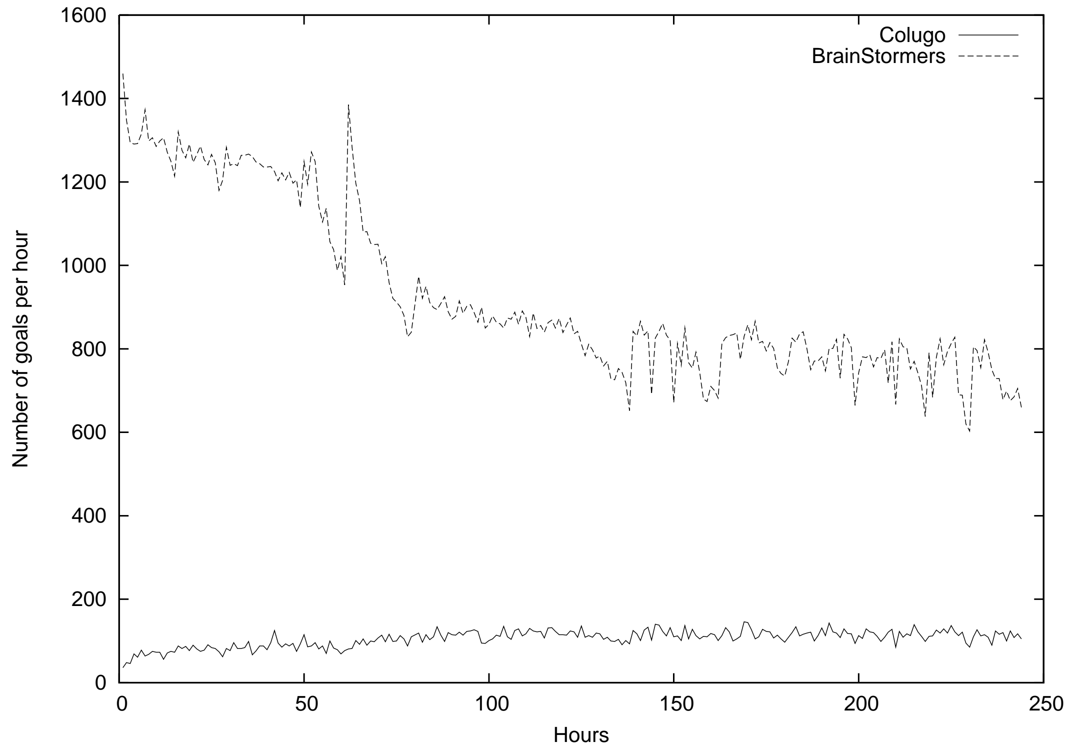

Figure 4.5: Exp. 7: Number of goals per hour

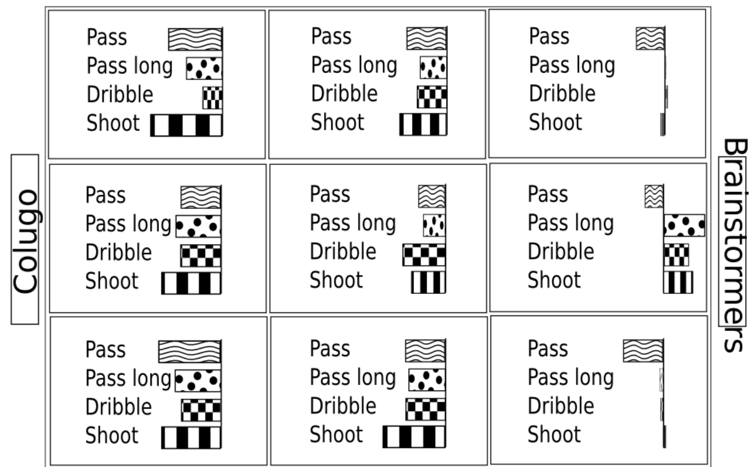

Figure 4.6: Exp. 7: Scores for the various actions for a player with the ball, by area

4.15.3 Conclusion

Long passes were a success, and consistently scored higher than the regular passes. They even inspired the unexpected technique of passing long right in front of the goal, to allow the teammate to save if the kick goes off in the wrong direction. However, the score is not especially good in spite of significant learning.

Chapter 5

Results

5.1 Introduction

This chapter will present a summarisation of the results of our experiments in Section 5.2.

5.2 Summary

Our first two experiments clearly showed that it is not feasible to directly apply reinforcement learning in the form of UCT to learn to play robot soccer in the RoboCup 2d setting, using only the most basic information about the world and the most basic actions. The number of involved variables leads to an exponential explosion; the learning module that keeps the information about the visited states ate up all available memory within one to two hours of starting.

Experiment 3 through 7 idolised a modified BrainStormers agent and the BrainStormers infrastructure for state definition and actions. This setup, although more limiting than the one from the first two experiments, worked to a satisfactory degree and we were able to perform learning in settings with both one and two opposing players. After some tweaking of state definitions and action sets in Experiment 3, 4 and 5 our agents performed well in the 2 vs 1 setting of Experiment 6 with a 50/50 ratio of goals scored even though our agents were not given enough time to train to master all situations, especially those where it was not close to the ball.

The results of Experiment 7 turned out to be a little disappointing considering the good results in Experiment 6. Our agents did not manage to play on par with the opposition when facing even odds in a 2 vs 2 match. After training for a long time, 4 times longer than in Experiment 6, our agents seemed to converge to a scoring ratio of 15/85.

Chapter 6

Software Evaluation

Here we will evaluate the software we ended up using in this project. Soccer server will be evaluated in Section 6.1, the BrainStormers code base will be evaluated in Section 6.2, and the UCT framework from (Holen & Marøy, 2007) will be evaluated in 6.3

6.1 Soccer Server and related tools

The RoboCup Soccer Server, or rcssserver as the package is called, was highly amenable. It was simple to compile and run, and it had no problems with modern 64bit GNU/Linux systems. The same went for the associated tools like the soccer monitor and log replay program. However, once in a while, the soccer server would crash. Out of 15 running instances, one might crash roughly every 8 hours. The learning system in general wasn't affected by the loss of servers, so fortunately this was too rare to disrupt the experiments.

6.2 BrainStormers

We did not expect the BrainStormers release to be very polished, since it's been a private project and has a very small target group. We were right in that. BrainStormers refuses to compile under recent versions of the GNU C compiler. When we started the project, the current version was GCC 4.2.2, but BrainStormers refused to compile with anything after GCC 3.3.6, and immediately crashed when run in 64bit mode.

Some parts of the code were very well commented, complete with ASCII art schematics. Other parts merely had either nothing or bits of German. It was quite difficult to work out the flow of control as it relies extensively on global variables. The actions seemed modular and easy to use in new behaviours, even if some had surprising quirks. For example, the module that made the agent run to a position would indicate when the action was complete, but the module that kicked the ball did not and required an enforced time limit to prevent infinite dawdling. However, some of the actions, such as the NeuroKick05 action for kicking the ball, did not appear to work as well when decoupled from the BrainStormers action selection code. This could cause occasional erratic kicking behaviour and missed goal opportunities.

We did not have to modify much of the BrainStormers code, we mostly added new code and left the existing code in place. There were however some small bugs that appeared. The worst one of these were an out of bound write to a multi-dimensional array when a player would accidentally run outside the field. This caused a stack smash several frames above the offending code, and thus it did not make the client crash until the program had run thousands of lines longer, making debugging exceedingly difficult.

All in all, once we finally got BrainStormers running, it was a fairly good base for our experiments.

6.3 Kerodon UCT base

We based our UCT program on Kerodon (Holen & Marøy, 2007), from our previous project. It was generically designed, but turned out to require a fair amount of re-engineering. Kerodon was written to only control one agent per simulation, and did not consider the addition of coaches to control the game. Still, we were able to reuse all of the UCT-related code even if the surrounding framework needed modifications, and this provided us with a tested platform.

Chapter 7

Project evaluation, conclusions and further work

In this chapter we will evaluate the project in Section 7.1, draw conclusions and answer the problem statement in Section 7.2 and share our views on what further work could be done on the projects subject in section 7.3.

7.1 Project Evaluation

The experiments we have had time to perform during this project has not provided much conclusive evidence for or against the viability of using reinforcement learning with UCT in robot soccer learning. Despite this we feel that our work has provided some interesting answers, interesting new questions and that it can serve as a good basis for further projects in the area. We will present our ideas for what further work can be done, along with how we would have continued our work if we had more time to invest in this project in Section 7.3

7.2 Conclusions

7.2.1 Answer to problem statement

Our initial experiments showed that it is not possible to directly learn to play robot soccer using reinforcement learning with UCT using the most basic state descriptions and actions on current hardware. Further experiments with a higher level of abstraction to state descriptions and actions resulted in significant learning but multi-agent planning did not emerge. This does not mean that it is unfeasible for multi-agent planning to be learnt. and further improvements to the experiment setting might be what it takes for planning to emerge.

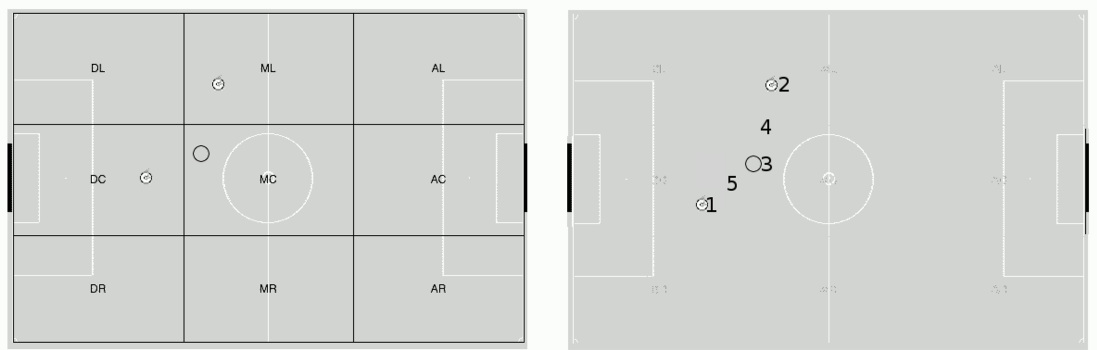

Figure 7.1: Zone based defence (left) and Player based defence (right). In this scenario, running the the centre of a zone is not relevant, but around players, the ball, or between a player and a ball might be.

7.3 Further work

This section presents our recommendations for further work that can be done in this area.

7.3.1 Player-based or zone-based defence

In our experiments, we used a zone-based defence where the field was divided into nine static zones. Analysing our experiment results we found that this had several problems. If you are going to use zone defence effectively you have to divide the field into smaller areas than we have done. With as large areas as we have you have a too big probability to end up too far away from the opponent you want to defend against if you move to the centre of the same area as him. If you try to run to an area between your goal and the area of your opponent you will end up even further from him, and not necessarily in his path. You can improve this situation by dividing the field into smaller areas, but then you will have to pay the cost of an exponentially larger state space.

A future project should consider a player-based defence instead, by defining dynamic key areas on the field, such as around opponents, between opponents and the ball, or between the ball and the goal. This would allow smarter movement options for the agents while keeping the state space small. Possible key areas are illustrated in Figure 7.1.

7.3.2 Colugo agent vs BrainStormers agent: not a fair match

The BrainStormers agents action selection scheme differs from the Colugo scheme in the way that BrainStormers often queue up several low level skills to be performed in series, and some of the skills are designed in such a way that they work very well as part of a series, but less well on its own. As an example we had problems with the performance of the kicking skill *neurokick05*. It had especially unpredictable behaviour if the ball was

not located in front of the player before kicking, but off to one side. This means that the skill performed well for the BrainStormers agent that used it after dribbling the ball a distance and having the ball exactly where it wanted it, but not so well for our agent that did not have the possibility to line up with the ball before kicking.

This coupling between usage and design of skills leads to the BrainStormers agent having an inherent advantage over other agents using the same skills in another context. Further work with our modified BrainStormers agent should include minor design changes to used skills to make them better suited for individual use, or introduce necessary helper actions for each skill so they can be used in the way they are designed to be used.

References

Auer, P., & Cesa-Bianchi, N. (2002). Finite time analysis of the multiarmed bandit problem. *Machine Learning*.

Gelly, S., Wang, Y., Munos, R., & Teytaud, O. (2006). Modification of UCT with Patterns in Monte-Carlo Go.

Guestrin, C. E. (2003). Planning under uncertainty in complex structured environments.

Holen, V., & Marøy, A. (2007). Reinforcement learning in Wargus.

Jensen, F. V., & Nielsen, T. D. (2007). *Bayesian Networks and Decision Graphs*. Springer.

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1995). Planning and Acting in Partially Observable Stochastic Domains.

Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo Planning.

Lai, T., & Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathemathics*.

Park, K.-H., Kim, Y.-J., & Kim, J.-H. (2001). *Modular Q-learning based multi-agent cooperation for robot soccer.*

Riedmiller, M., & Gabel, T. (2007). Brainstormers 2D Team Description 2007.

Schaeffer, J., & Plaat, A. (1997). Kasparov versus Deep Blue: The Re-match. *ICCA Journal*.

Schmidhuber, J. (1996). A general method for multi-agent reinforcement learning in unrestricted environments. *Adaptation, Coevolution and Learning in Multiagent Systems: Papers from the 1996 AAAI Spring Symposium*.

Schwab, B. (2004). *Ai Game Engine Programming*. Charles River Media.

Stone, P., & Veloso, M. (1997). Multiagent Systems: A Survey from a Machine Learning Perspective.

Watkins, C. J. C. H. (1989). Learning from Delayed Rewards.

Yang, E., & Gu, D. (2004). Multiagent Reinforcement Learning for Multi-Robot Systems: A Survey.

Index

absorbing state, 11
abstract, iii
action selection, 14
actions, 10
AIs
  competitive, 2
  contemporary soccer games, 24
  entertaining, 2
  in games, 1
approach, 4
architecture, 25
  BrainStormers, 21
  Colugo, 25
audio, 9

BrainStormers, 21
  architecture, 21
  modifications, 27

chapter overview, 4
coaches, 10
Colugo
  architecture, 25
competitive AI, 2
conclusions, 53
confidence interval, 15
cumulative reward, 11

deliberative agents, 19

entertaining AI, 2
evaluation, 51
  BrainStormers, 51
  Kerodon UCT base, 52
  project, 53
  soccer server, 51
Experiment 1
  description, 30
  results, 31
Experiment 2
  description, 31
  results, 32
Experiment 3
  description, 33
  results, 35
Experiment 4
  description, 36
  results, 38
Experiment 5
  description, 39
  results, 40
Experiment 6
  description, 40
  results, 41
Experiment 7
  description, 43
  results, 44
experiments, 29
  introduction, 29

further work, 54

global perspective, 19

hearing, 9

introduction, 1
introspection, 10

leagues, 5
local perspective, 19

Markov Decision Process, *see* MDP
Markov property

soccer server, 18
MDP
constraints, 12
definition, 11
soccer server, 17
solving, 12
modular Q learning, 23
multi-agent environments, 19

offline coach, 10
online coach, 10
overview, 4

Partially Observable MDP, *see* POMDP
physical properties, 7
physics, 7
play field, 7
players
actions, 10
hearing, 9
introspection, 10
vision, 9
policy iteration, 12
POMDP, 12
soccer server, 17
prestudy, 5
problem statement, 3
project evaluation, 53

Q learning, 13

reactive agents, 19
reinforcement learning, 13
constraints, 15
Results, 49
results
summary, 49
reward
cumulative, 11
RoboCup, 5
humanoid, 7
leagues, 5
middle size, 6
simulation, 5, 7
simulation environment, 7
small size, 6
standard platform, 6

simulation, 7
physics, 7
players, 9
time, 9
simulation environment, 7
state of the art, 20
stationary, 18

UCB1, 15
UCT, 15, 16
implementation, 26

value iteration, 12
vision, 9

world, 7

zone-based defence
conclusion, 54