Action Selection and Action Control for Playing Table Soccer Using Markov Decision Processes

Master Thesis

Dapeng Zhang

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Grundlagen der Künstlichen Intelligenz
Supervisor: Prof. Dr. Bernhard Nebel

April 2005

Acknowledgments

Thank all people who helped me complete this master thesis. Especially thank Bernhard Nebel for being the supervisor of this thesis. Thank Thilo Weigel for helping me go through this thesis. Thank Alexander Kleiner for the very helpful suggestions on the Markov Decision Processes, Reinforcement Learning, and related readings. And thank Zeno Adams and Daniela Wack for correcting my English writing. I appreciates the helps you did for this master thesis.

Declaration

I hereby declare that I wrote this thesis on my own, only making use of the sources mentioned.

April 2005, Dapeng Zhang

Abstract

StarKick is a commercially available table soccer robot which challenges even advanced human players. However, the available set of actions for StarKick is limited and the way of selecting the actions is not flexible enough for incorporating more elaborate actions.

In the context of this thesis, new actions for stopping and dribbling the ball are developed. Stopping is achieved by locking the ball between the playing surface and a playing figure. Dribbling makes the ball rolling at a controllable speed within the reachable area of the playing figures of one rod. By these new actions, the ball can be deliberately passed and stopped.

To decide, which action should be taken in a given situation, an action selection scheme using Markov Decision Processes (MDPs) and reinforcement learning is developed. In order to reduce the state space, the basic actions are combined to more complex actions and the MDP is structured into four modules. Each module contains a set of states, and the actions that are applicable in these states. A simple reinforcement learning algorithm is implemented in the MDP framework. The transition probabilities are updated by counting. These updates are spread by policy iteration algorithm during a game.

A series of experiments are carried out in real table soccer games. These experiments show that the newly developed actions are robust, the MDP model works fine, and the reinforcement learning improves the performance of StarKick in a simplified game. The attempt of making the reinforcement learning work in the whole game seems too tedious to be finished in real games.

Contents

1 Introduction **1**

2 KiRo and StarKick **3**
2.1 Vision System and World Model 4
2.2 Previous Solutions of Action Control and Action Selection 6

3 Action Control of StarKick **11**
3.1 StopActions 12
3.2 KickActions 14
3.3 PassActions 15

4 Theoretical Background **19**
4.1 Markov Decision Processes 19
4.2 A Simple Reinforcement Learning Algorithm 22
4.3 Dyna Architecture 23

5 Action Selection of StarKick **27**
5.1 The MDP Model 28
5.1.1 States 29
5.1.2 Actions 32
5.1.3 Four modules of the MDP model and the transitions 34
5.1.4 Rewards 38
5.2 Opponent Models 39
5.3 Implementation of the Reinforcement Learning 40
5.3.1 Learning from the *Blocked* Model 40
5.3.2 Learning from Dynamic Model 40

6 Software Implementation **43**
6.1 Environments 43
6.2 Implementation of Action Control 43
6.2.1 Difficulties and Solutions 44

6.2.2 Action Games 45
6.3 Implementation of Action Selection 46
6.3.1 Control Program for the MDP model 47
6.3.2 Extra Opponent Models for Implementation 47
6.3.3 Transient States 48
6.3.4 Implementations of Opponent Models 49
6.3.5 Graphic User Interface 52

7 Experiments **53**
7.1 Evaluation of the Basic Actions 53
7.2 Evaluation of the MDP and RL 57

8 Conclusion **63**
8.1 Summary 63
8.2 Future Work 64

A Phrases **67**

B Contents of the Appendent CD **69**

Chapter 1

Introduction

Table soccer is a popular game in bars or other amusement places. It could be a game played in a way for relaxation as a hobby, or as a sport to compete in championships. There are two sides in a table soccer game, one playing against the other. KiRo is a table soccer robot, which controls one side, competing with human beings at the other side.

KiRo was firstly developed in the Freiburg University. It successfully challenges human beings in table soccer games. A patent [6] was applied for KiRo because of its commercial value. A German company bought the license and developed the second generation table soccer robot named StarKick based on KiRo. StarKick has been commercially available since January 2005 [10]. And KiRo becomes a successful example of combining scientific research with commercial benefits.

Compared to KiRo, StarKick is characterized by significant improvements of the hardware. The mechanical system is better and more robust. Extra sensors are mounted to observe the angles of the rods. The vision system achieves an average deviation of around 3mm on the ball recognition. These optimizations facilitate more complex actions, which should make StarKick behave more intelligently in the games.

Action control and action selection in the noisy, dynamic, and not completely predictable environments are regarded as one of the central problems of intelligence in robotics. The basic actions of a robot are given in its action control. Action selection is the selection of these basic actions in different situations. In case of StarKick, it acts in the environments by basic actions such as *Kick*, *Block*, and *Clear*, which belong to the action control. A particular action is selected according to the information of the ball and rods, which belong to the action selection. For example, when the ball is controlled by the opponent, *Block* is selected in the action selection [11]. In other words, action control and action selection determine the behaviors of StarKick in the games.

Markov Decision Processes and Reinforcement Learning are popular research areas in robotics. Markov Decision Processes (MDPs) are one of the approaches that can be used to solve the problem of the action selection. MDPs make the sequential decisions with high-level mathematical models [2], and these decisions become the selections of the actions when the MDPs are used in the action selection. Reinforcement Learning (RL) is to learn the feedbacks from the performed actions, and use these feedbacks for the next decision. RL can be implemented in the MDP framework.

In this thesis, basic actions such as *Lock* and *Pass* are developed. *Lock* makes the ball stuck between the playing surface and the playing figure. *Pass* makes the ball rolling at a controllable speed within the reachable area of one rod. Furthermore, an action selection method is developed using Markov Decision Processes (MDPs). In order to adapt the actions to the MDPs, basic actions are combined to more complex actions which can be eventually used in the MDPs. For example, *Lock* and *Pass* are combined to *Dribble* actions. Since some actions are not applicable in some states, the MDP implemented in this thesis consists of four modules. Each contains a set of states, and the actions that are applicable in these states. In addition, a simple Reinforcement Learning algorithm is implemented in the MDP framework. The transition probabilities are updated by counting. These updates are spread by policy iteration algorithm during a game.

This thesis is structured as follows. Chapter 2 gives a general introduction to KiRo and StarKick, focusing on the architecture of KiRo, the difference between the hardware of KiRo and StarKick, as well as action control and action selection mechanisms. Chapter 3 depicts the new actions implemented on StarKick. In addition to previously developed *Kick*, *Clear*, and *Block* actions [11], *Lock* and *Dribble* actions are developed. These basic actions create a new basis for the action selection in this thesis. Chapter 4 introduces the theoretical background, including Markov Decision Processes, Reinforcement Learning, and Dyna Architecture – a full reacting system with planner [8]. Chapter 5 shows how the action selection of StarKick is modeled using MDPs. As the behaviors of the opponents cannot be observed directly in StarKick, two type of human behavior are proposed. The reinforcement learning in Chapter 5 learns the transition probabilities to play against these human behaviors. Chapter 6 is organized according to the steps to implement the algorithms discussed in the previous chapters, from simple to complex ones, containing the implementation of actions, MDPs, and RL algorithms. Chapter 7 shows the experiments, and finally, chapter 8 draws the conclusion.

Chapter 2

KiRo and StarKick

KiRo and StarKick are two generations of a table soccer robot. KiRo is the first generation developed at the Freiburg University by mounting mechanical system and the overhead camera on a normal game table. It can control one side of the table soccer game, playing against human beings at the other side. As the second generation based on KiRo, StarKick is a robust product that can be served as an automaton like other game machines in bars. Instead of the university, a German company developed StarKick. Figure 2.1 shows the appearances of KiRo and StarKick. The mechanical systems as well as the camera of StarKick are hidden in the body of the automaton under the playing surface.

Although the hardware systems are quite different, StarKick and Kiro use the same software. The architecture of the software is shown in Figure 2.2. There are two big parts in the architecture, the upper Graphic User Interface (GUI), and the lower software components. There are four components in the lower part. "Action control" and "action selection" define the behaviors of StarKick in the games. "World model" is the environments from the robot points of view. The world model of StarKick contains the position and speed of the ball, as well as the positions and angles of the rods. The "vision" processes the frames from the camera to get the environment information. At the beginning, world model is updated according to the observed data by the sensors and camera. Then, action selection chooses an action according to the world model. Finally, action control sends commands to motor to perform the selected action. This architecture is given by Thilo Weigel [11].

The rest of this chapter introduces the vision system, world model, and the previous approaches on action control and action selection.

(a) (b)

Figure 2.1: The pictures of (a) KiRo. (b) StarKick.

2.1 Vision System and World Model

The vision system of StarKick is different from KiRo. Figure 2.3 (a) shows one image from the camera of KiRo, and (b) shows one image from the camera of StarKick. Both images have a resolution of $384 * 288$ pixels. The camera observes the game, and sends the images as frames to the computer. By processing each half frame separately, both of KiRo and StarKick achieve the frame rate $50Hz$. [10] These discrete frames refresh the world model every $20ms$. KiRo gets the information of the ball and rods by image processing. [12]. There is a problem, which is called "hidden ball" problem, existed in the vision system of the first generation table soccer robot KiRo. If the ball is under a playing figure, it cannot be observed anymore from the overhead camera. In the second generation, the camera of StarKick is set to avoid this problem because it observes the semi-transparent playing surface from below in the body of the machine. As shown in Figure2.3(b), the camera view of StarKick does not include any information of rods. They are given by the motor encoders and sensors, which are more accurate than the data of KiRo. Although StarKick does not have the "hidden ball" problem, it has no information of the rods of the human opponent, which is nevertheless important for the action selection. Figure 2.4 shows the visualizations of the world models of KiRo and StarKick. In the figure, we can see that StarKick has no information of the opponent rods.

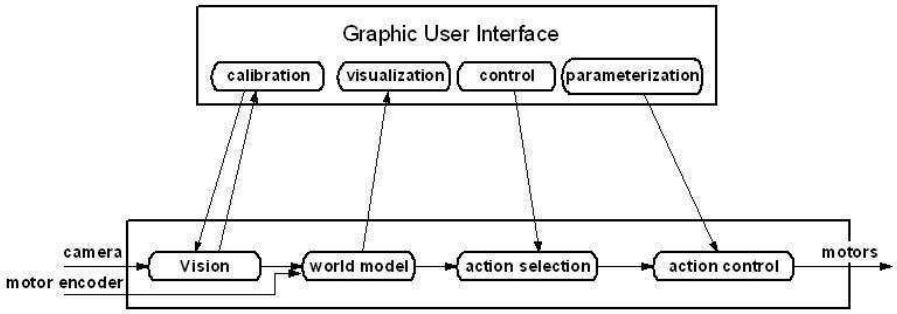
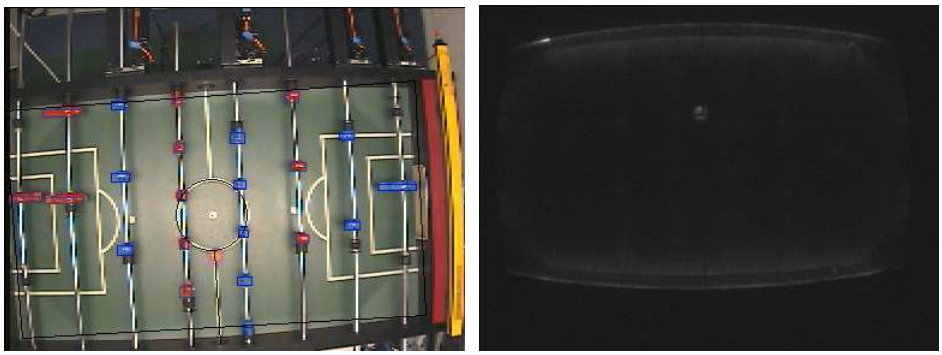


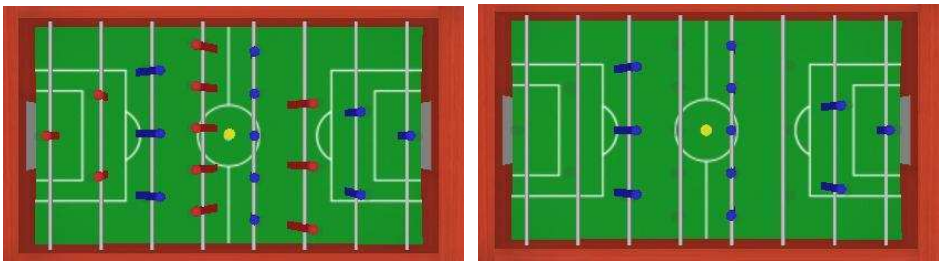
Figure 2.2: Software Architecture of KiRo.



(a) (b)

Figure 2.3: Camera view of (a) KiRo and (b) StarKick.

The world model has a two dimensional coordinate system of the playing surface shown in Figure 2.5. The zero point of this coordinate system is the center of the table. X axis is from the goal of StarKick to the goal of human opponent. Y axis is parallel to the axis of the rod. Negative and positive directions which are widely used in this thesis means the directions along positive or negative x axis. For example, in Figure 2.5, the direction from left to right along X axis is the negative direction, whereas the direction from right to left is the positive direction. The position of the ball in the camera view is mapped to this coordinate system which has the same dimensions as a real game table, 1200mm along x, 680mm along y.



(a) (b)

Figure 2.4: The Visualizations of World Model of (a) KiRo and (b) StarKick

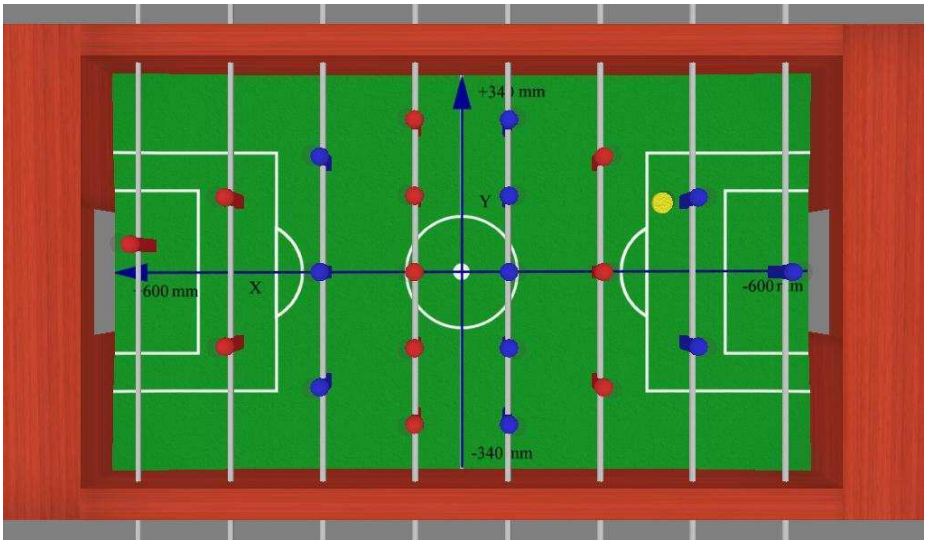


Figure 2.5: Coordinate System.

2.2 Previous Solutions of Action Control and Action Selection

Before this thesis, two other action control and action selection systems have been implemented on KiRo.

The first approach has the action control system described in Figure 2.6. In the description, the "forward" direction is positive direction, and the "behind" means the negative direction. There are a lot of experiences which are valuable for this thesis. The most important three ideas are used by the action control of this thesis. These ideas are suitable to different situations. When the ball can be reached by two playing figures of the same rod, a small change in y coordinate of the ball would make the robot switch the playing figures.

However, the position of the ball is actually changing within 3mm all the time, even if when the ball is still because of the noise of the camera. This noise makes KiRo oscillate between two applicable figures. The first idea is that the action control system prefers the selected rod in the blocking to avoid this oscillation. When the ball is far away to a rod, It is not necessary to move the rod very fast to a position. The second idea is to consider how urgent the situation is to avoid hectic movement. The third idea is a effective blocking strategy on how to defend. This strategy is used in the commercial StarKick even today.

- DefaultAction move and turn rod to default home position.
- KickBall rotate the rod by 90 degree to kick the ball forward.
- BlockBall move the rod so that a figure intercepts the ball.
- ClearBall move to the same position as BlockBall but turn the rod in order to let the ball pass from behind.
- BlockAtPos move the rod so that a figure prevents the ball passing at a specific position
- ClearAtPos move the rod to a specific position and turn the rod in order to let the ball pass from behind.

Figure 2.6: Action Control of KiRo [11]

The action selection model in the first approach works in a reactive way shown in Figure 2.7. It is a simple decision tree, in which KiRo selects an action in the real-time manner by some simple predicates. For instance, as soon as the ball is kick-able, KiRo will select *Kick* action. [11].

The second approach employs the similar action control but a different action selection method. In this approach, decision-theoretic planning is used for the action selection. The planning is started at a game state s, at which the ball is controlled by KiRo. Rooted at this state s, a planning tree is constructed to make the action selection. The first step to construct the planning tree is to take possible actions in the state s as branches, as shown in Figure 2.8(a). The selection of the actions is to choose a branch in the planning tree. The second step is to formalize the possible reactions of the human opponents, as shown in Figure 2.8(b). In the second step, the reactions of the human opponents are assumed to be independent to the taken action in the first step. This holds in the real table soccer game

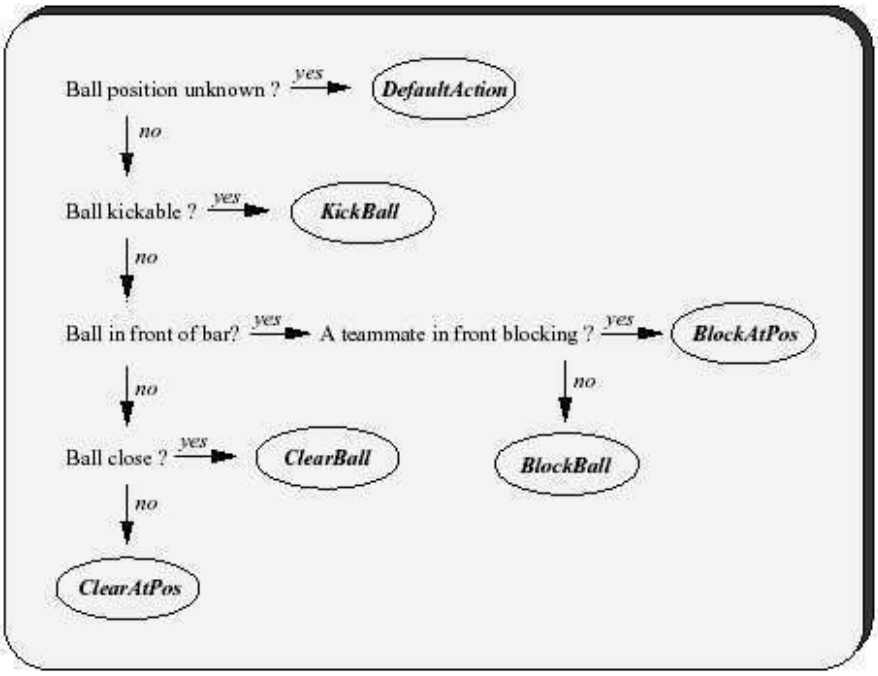
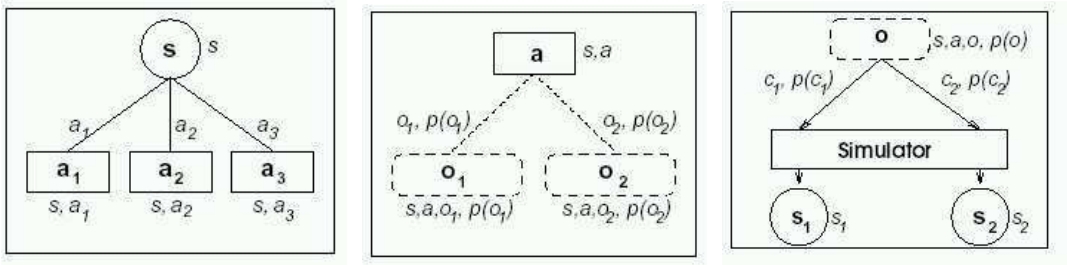


Figure 2.7: Action Selection of KiRo [11]

because KiRo performs an action within half a second, during which human opponents can not yet react to the taken action. Every possible reactions to the taken action is associated with a respective probability. A learning algorithm is employed to obtain and update these probabilities. In the third step, a software simulator is used to simulate the consequences of the actions of KiRo and the reactions of the human opponents, as shown in Figure 2.8(c). It is configured and run several times on the different combinations of actions and reactions to reveal the rewards of each situation. The action, which yield the maximum reward value, is going to be selected finally. Although our simulator has considered the frictions, bounces, and collisions between the ball, walls, and playing surface, a game in the software simulator is very different from a real table soccer game. However it can match the real game to some extent. In the third step, the software simulator is configured according to the observations of the real game. It can match the situations of the real table soccer game a short while after it is configured. In the decision-theoretic planning approach, the simulor is run until another planning state is reached. The whole planning tree can be constructed by repeating the above three steps. One layer of the planning tree is made up with this three steps. The planning reaches the end if the ball goes into one of the goals in the planning tree. [9]

In a PC with a $2.6GHz$ CPU and $512M$ memory, the depth of the planning tree can reach two layers within one process cycle of KiRo. A heuristic function is therefore em-



(a) (b) (c)

Figure 2.8: One Iteration of the Planning [9]

ployed to estimate how good the situation is, so that the branches of the planning tree can be estimated even if the goals are not reached. [9] In other words, this approach employs an open loop planning. Although it may reach the end if there is a faster enough computer, the simulation steps used in this approach may not match the situations of a real game after several steps of the planning. The second approach outperforms the first one in the software simulator, but not in the real table soccer games.

Chapter 3

Action Control of StarKick

Action control is crucial for the behavior of StarKick because it is the basis of action selection. Although there is not much theoretic stuff in the background of the action control, the improvements of it actually motivates this thesis. The new actions developed in the action control act the role of making the final game interesting.

Actions are classified into four different sets according to their preconditions. These sets are *StopActions*, *Clear*, *KickActions*, and *PassActions*, as shown in Figure 3.1. The preconditions of the actions are important because they define the situations in which the actions are applicable. Action selection only works when there is at least two actions applicable. In case of StarKick, action selection chooses an action from one of these four action sets in the situated environments. Clear action shown in Figure 3.1 is not been changed in StarKick, other actions are newly developed actions, which are organized as action sets and discussed one by one in the later sections.

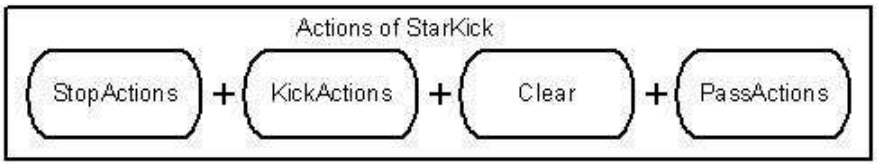


Figure 3.1: Action Sets of StarKick

The actions in this chapter are all atomic actions. In other words, although each action is the combinations of a sequence of turnings and movings, it is regarded as an un-dividable atom. The alternative would be having a very simple action control part with turning or moving as one action. Then it comes to the problems that the number of actions becomes

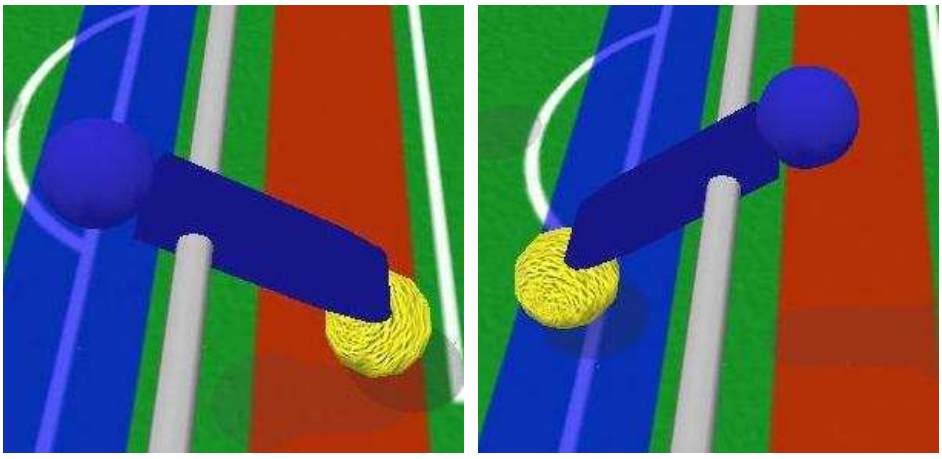
infinite because the turning and moving are in the continuous coordinate system. Actually, in this thesis, these atomic actions are combined to more complex units which can be eventually used in the action selection.

3.1 StopActions

There are three actions in the *StopActions*. They are *SoftBlock*, *LockFast*, and *LockSlow*, The function of them is to get the ball under control. *SoftBlock* is a passive action to defend. It is widely used when the ball is under the control of a opponent. Whereas *LockFast* and *LockSlow* can actively lock the moving ball.

The action *SoftBlock* inherits the features of *BlockBall*. As an additional feature, the peak motor current of the rod, which determines the motor strength to hold the turning angle, is set to a very low value, so that when the ball is bounced back by a playing figure, it will be slowed down significantly.

The ball can be actively locked in negative and positive lock bands, which are shown in Figure 3.2. Negative lock band of the defender is shown as red region in the figure; positive lock band is shown as blue region. They are positive and negative in x direction with respect to the rod axis of the defender.

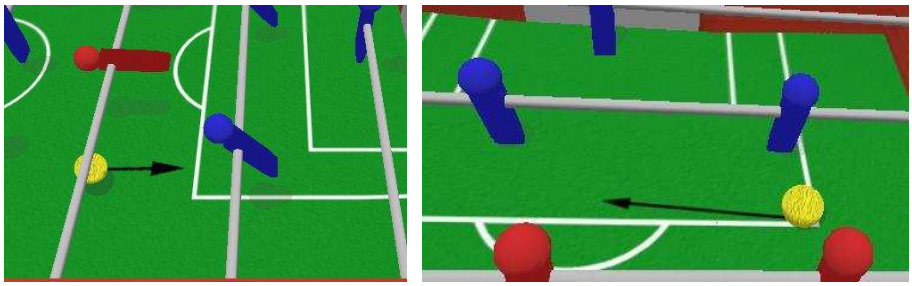


(a) (b)

Figure 3.2: (a) Negative Lock Band (b) Positive Lock Band

LockFast is developed to lock the ball moving along the x direction. Figure 3.3(a) shows this situation. It predicts the point of time at which the ball is rolling into a lock band, and presses the ball to the playing surface to stop it. The speed vector of the ball

is used in the calculation. *LockSlow* is developed to lock the ball that is in the control range and moving roughly along the rod axis. It assumes that the speed of the ball along x direction is very slow. Figure 3.3(b) shows this situation, which is typically happened after a passing. In order to reduce the turning time of the rods, at the very beginning, both LockFast and LockSlow turn the rod to a place at which it is ready to do the "pressing".



(a) (b)

Figure 3.3: The situations of (a) *LockFast* and (b) *LockSlow*

A very important feature of *LockSlow* action is that it can turn to any angle without disturbing the rolling of the ball. This is difficult because there is a bit of oscillation around the target angle when the rod is turned to it, as shown in Figure 3.4. Although the direction of the turning is always set in the way that the figure will not touch the ball, only the oscillation is enough to damage any intended action. In order to solve this problem, the figure is firstly moved for a short distance; then it is turned to the target angle; finally the figure is moved back.

Both *LockFast* and *LockSlow* need to know where the exact place to do "lock" is. Unfortunately, the coordinate system is not accurate enough to support the lock actions. The angle of the rod at this lock place, as well as the ball position with respect to the rod axis, are measured by hand for all different figures in negative and positive directions. Instead of hand measuring them, these parameters should be adapted in a learning process by observing the results of locking. This part of work is not included in this thesis due to the limitation of time.

As introduced in Chapter 2, the continuous world is observed every $20ms$. From another aspect, there is a chance to send commands to the motors every $20ms$. There exists the situation that it is too early to send the turning command and lock the ball in one cycle, but too late when the next chance comes after 20ms. In order to solve this problem, the turning speed and acceleration of the rod is set to lower value so that the turning takes more time, and the lock command can be sent at the first chance. The activity diagram of *LockFast* is shown in Figure 3.5.

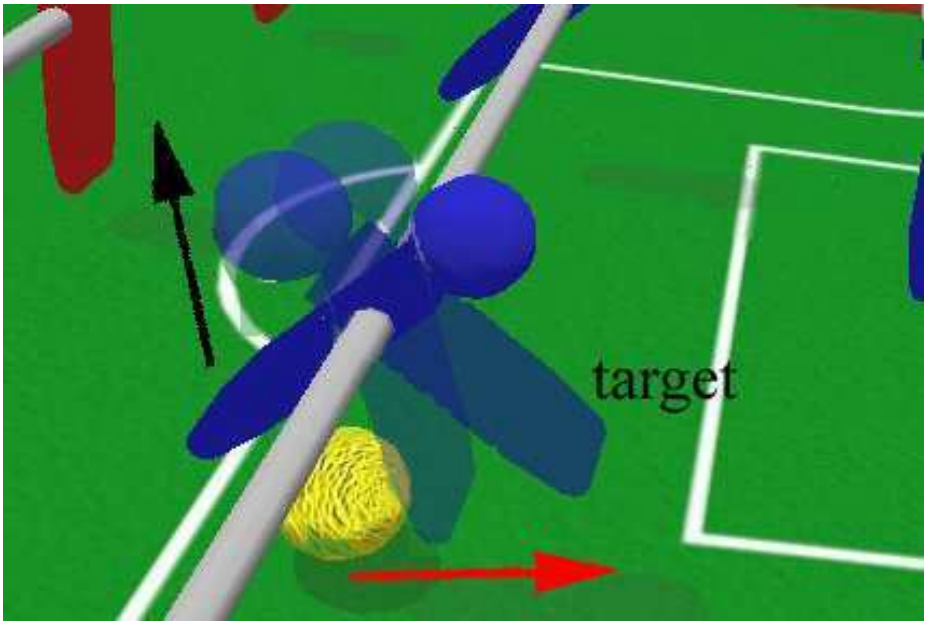


Figure 3.4: the oscillation of the rods after a turning

3.2 KickActions

There are six actions in this action set. They are *KickFastForward*, *KickFastLeft*, *KickFastRight*, *KickSlowForward*, *KickSlowLeft*, and *KickSlowRight*. From another aspect, *KickActions* have two parameters, the directions (forward, left, and right) as shown in Figure 3.6 and the speed (fast and slow). The speed of the kick is implemented by setting the turning speed and the motor current of the respective rod.

KickActions are implemented in a robust way so that a kick intention can be expressed in whatever situations. Of course a "kick" is only applicable when the ball is kick-able; however the *KickActions* in this thesis integrated the strategies when the ball is not kick-able, so that the action selection described in this thesis needn't consider too much of the environments, but only express a kick intention. When the ball is far away, the effects of the kick actions are same with *BlockBall*. When the ball is under the control but not kick-able, kick actions turn the figure to the angle ready for a kick and wait. The turning here is encoded to avoid disturbing the rolling ball as in *LockSlow*. *KickActions* considers the situation that a performed kick action did not touch the ball. In this situation, any action in *KickActions* can be called continuously without any initial stage, and the turning of a kick

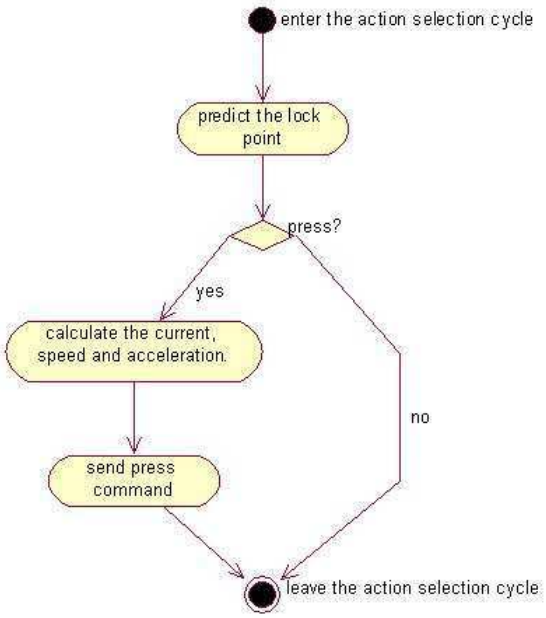


Figure 3.5: The activity diagram of the *LockFast*

would have an arbitrary angle from 90 degree to 360 degree.

The left and right kick are implemented by moving the kicking figure to have a $\pm 10mm$ different along y axis with the ball. Obviously, a kicking with same y position with the ball should be a forward kick. However, because of the noisy environments and the indefinite effects of actions, when the ball is moving or even it is still, it is hard to have an exact left, right, or forward kicking. Choices for different kicks are made in action control, so that action selection can decides which one is better in a real game.

3.3 PassActions

There are four ways to pass a still ball, which are *PassLongBack*, *PassShortBack*, *PassParallel* and *Touch*, as shown in Figure 3.7. Multiplying the four ways with two pass directions (left and right), eight actions are developed in this action set.

All these actions assume the precondition that the ball is still and under control. *Touch* actions are applicable when the ball is still but not locked, which is simply called "still" in this thesis, whereas others are applicable when the ball is locked. Because lock is an active action to stop the ball while still is a passive result, in a real game, "lock" would

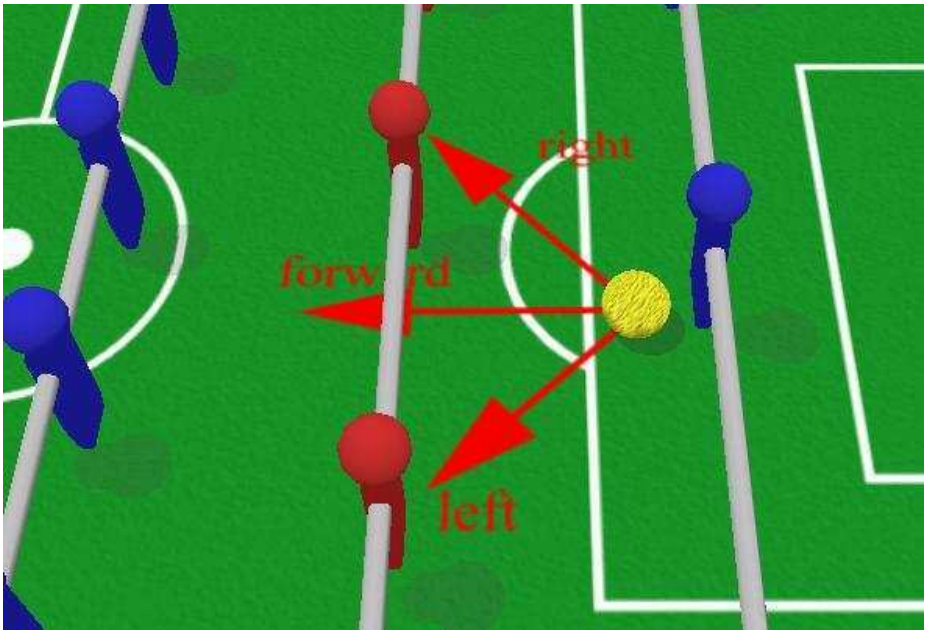
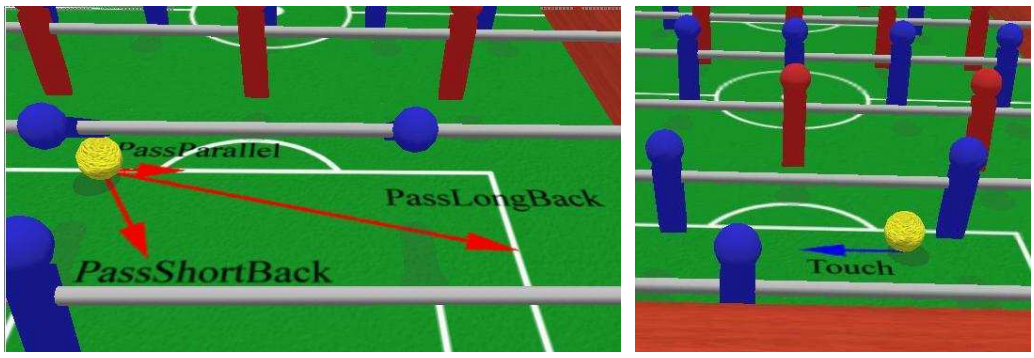


Figure 3.6: The three directions of the *Kick*

be more likely to happen than the still. *Touch* actions are developed to make the "lock" actions applicable in the still situations.

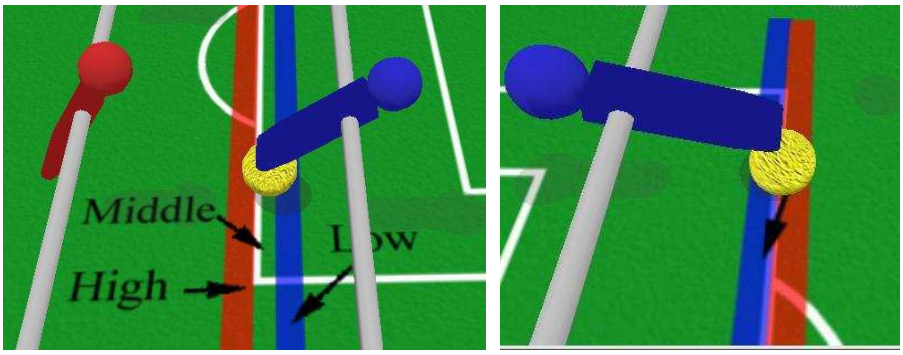
PassLongBack is applicable when the ball is locked. Here "back" is with respect to the rod axis. In particular, if the ball is in the negative lock band, it is passed towards the positive lock band, and if the ball is in the positive lock band, it is passed towards the negative lock band. The motor current of turning used in the passing has three predefined values, which depend on the distance between the locked ball and the axis of the rod. Since the ball should be passed back, higher turning motor current is used to prevent the ball from slipping when the ball is further away, and lower motor current is used to avoid kicking when the ball is closer. These situations is shown in Figure 3.8 (a).

In *PassLongBack*, the figure which locked the ball is moved to the left or to the right for 110mm to achieve the passing with low velocity and acceleration. The distance of 110mm is the distance comes from the experiments that a rolling ball behind the passing figure would not touch the figure again before the action selection makes the next decision. *PassLongBack* itself doesn't check whether or not the moving trial is free because the moving is in low velocity and acceleration. Even if one figure reaches the end of its moving range, the consequence will not be so serious, and the effects of this action may still keep positive.



(a) (b)

Figure 3.7: (a) *PassShortBack*, *PassLongBack*, *PassParallel*, and (b) *Touch*



(a) (b)

Figure 3.8: (a) The motor current of the *PassLongBack* and (b) the situations of the *PassParallel*

PassShortBack and *PassParallel* have the same precondition as *PassLongBack*. In *PassShortBack*, very low motor current and very slow moving speed are used so that the figure which locks the ball slips down from the top of the ball, and the ball is passed back slowly with the angle nearly vertical to the rod axis. *PassParallel* passes the ball with medium speed, but raise the figure before the figure's edge is slipped down from the ball. The ball is supposed to continue to go along the moving direction after the figure is raised. In practice, one performance of *PassParallel* will make the ball a little bit away from the axis of the controlling rod. If the ball is too far away, the same passing as before will cause a lost of control. Therefore, *PassParallel* checks the distance between the axis of the rod and the ball. If it is too much, raising the figure is delayed for a bit, so that the ball is passed back a little to avoid loosing control. These situations are shown in Figure 3.8(b). Because the ball is going parallel in the lock range, there are chances to lock it again after passing. *PassParallel* is the smallest passing action which determines the accuracy of the action.

Touch action has different preconditions assuming that the ball is still and under control. No other passing actions are applicable because they assume the precondition of the ball being locked. In the *Touch* action, firstly the rod is moved and turned to make the still ball between two figures, as the turning in *LockSlow*, the turning here also avoids disturbing the ball. One figure is moved slowly towards the ball until one of its sides touches the ball. The angle of the rod is calculated in the way that the touching point will always be at the centre of the ball. In principle the ball should go in the moving direction, parallel to the axis of the rod. However the ball always moves away a little because of the noise. The moving of the figure is slowly so that the ball is not likely to be very fast, therefore the ball can be locked or kicked after *Touch*.

Chapter 4

Theoretical Background

This chapter gives a brief explanation on the theories of Markov Decision Processes (MDPs), Reinforcement Learning (RL), and Dyna Architecture. These theories are a basis for the better understanding of the action selection approach developed in this thesis. MDPs are explained in Section 4.1. The definition of MDPs and the policy iteration algorithm are given briefly. A simple RL method is depicted in section 4.2. Since RL is a very wide topic in artificial intelligence, the RL introduction in this chapter is limited to the RL in the context of MDPs with finite search space. Dyna architecture is given in Section 4.3, in which MDPs and RL are organized to improve the reactive execution. The action selection approach developed in this thesis can be regarded as an implementation of the Dyna architecture.

4.1 Markov Decision Processes

The work of Moriz Tacke [9] has been introduced in Chapter 2.2, in which decision-theoretic planning is used to make sequential decisions for the action selection of playing table soccer. As a branch of decision-theoretic planning, MDPs are suitable to make sequential decisions with a compact mathematical model under the conditions of uncertainty. In this thesis, the uncertainty of the MDPs is limited to the effects of the actions. Partially Observable Markov Decision Processes (POMDPs) can handle the uncertainty in the states [1], which are not relevant to this thesis.

There are always multiple and conflicting objectives in a MDP model [1]. For example, scoring against the human opponents and being scored by the opponents are two conflicting objectives in a table soccer game. These objectives are always modeled as the goal states in the MDPs.

MDP is defined as a tuple $\langle S, A, T, R \rangle$, as shown in Figure 4.1. It can be viewed as an stochastic automata where the actions have uncertain effects and induce stochastic transitions between states. [1] MDPs accommodate the probabilities of transitions, so that a transition, which is started at a state-action pair and ended at another state, is expressed as a respective probability.

- A state space S; a set $G \subseteq S$ of goal states.
- Actions $A(s) \subseteq A$ applicable in each state $s \in S$.
- Transitions probabilities $P_a(s'|s)$ for $T(s, a, s')$, where $s', s \in S; a \in A$.
- Reward function $R(s)$, where $s \in S$.

Figure 4.1: The definition of the MDPs [2]

If we express all situations as the states S defined in Figure 4.1, the *policy* of MDPs can be defined as a function $\pi : S \rightarrow A$ [1]. In other words, the policy of MDPs assigns each possible state an applicable action. It is a solution of MDPs. The system will perform the action $\pi(s)$ whenever it finds itself in state s. The policy mechanism of MDPs satisfies the *Markovian assumption* in the sense that the selection of the actions are independent of the system history.

The system gets the rewards when it enters a state, which is given by the rewards function $R(s)$. If we take the sum of the reward obtained by executing π as the value of a policy $\pi(s)$, the *optimal policy* $\pi^*(s)$ is defined as the best policy which achieves the maximum rewards. Solving MDPs is to find the optimal policy

Since the actions are the bridges between the states in MDPs, we need to consider not only the immediate rewards given by the reward function $R(s)$ but also the potential rewards which are going to be obtained by executing the actions. The *utility* $U(s)$ of a MDP state is defined as a value that gives a measurement on how good the state s is, which is the left side of the Bellman Equation. The Bellman Equation is shown in Equation 4.1. The parameter γ in this equation is a factor to discount the future rewards. It is always set to a positive value smaller than one.

$$U(s) = R(s) + \gamma \max_a \sum_{s'} (T(s, a, s')U(s')) \tag{4.1}$$

Value iteration is an algorithm to solve MDPs, or search π^*. Utilities for all states can be obtained by iterately applying the Bellman Equation on each state in the state space. The actions which achieve maximum utilities are found out and finally consisted in the optimal policy π^*. Value iteration algorithm for calculating the utilities of the states is shown in Figure 4.2.

- REPEAT
- $\delta \leftarrow 0$
  - For each state $s \in S$, calculate its new utility U'(s) using Equation 4.1
  - if $\delta <\mid U'(s) - U(s) \mid, \delta \leftarrow \mid U'(s) - U(s) \mid$
- UNTIL δ is small enough.
- return U.

Figure 4.2: Value iteration algorithm for calculating the utilities

Comparing with Value Iteration, Policy Iteration can find the optimal policy before the utilities are converged. It is proved to be more effective in practice. The policy iteration algorithm starts with a random policy, and repeats the following two steps until there are no improvements made in the policy. [1] The policy algorithm proceeds in the way shown in Figure 4.3.

- *Policy Evaluation:* Instead of using the action which achieves maximum utility in the Bellman Equation, use the current policy to iterate on the states, until the utilities are converged.
- *Policy Improvements:* calculate a better policy by apply utilities calculated in the last step.

MPDs provide a framework for RL. The background information on RL is going to be given in the next section.

- Let π' be any policy on S
- While $\pi \neq \pi'$ do
  - $\pi = \pi'$
  - Iterately update the utilities using Equation 4.1 but only consider the fixed policy π.
  - For each $s \in S$ if there exists an action a' which is applicable on s and yields to a bigger utility value than $\pi(s)$, $\pi'(s) \leftarrow a'$
- return π.

Figure 4.3: Policy iteration algorithm [1]

4.2 A Simple Reinforcement Learning Algorithm

Reinforcement Learning (RL) refers to a class of learning tasks and algorithms in which the learning system learns an associative mapping, $\pi : X \rightarrow A$ by maximizing a scalar evaluation (reinforcement) of its performance from the environment (user) [4]. In this thesis, the RL is implemented in a MDP model with finite search space. This section only gives the background information on the RL under this restricted settings. The general information on RL can be found in many readings such as [3] and [4].

When the search space is not very large, the transition model $T(s, a, s')$ in MDPs can be denoted by a table named *transition table*, in which each cell records the respective probability of the occurrence of the resulted state s' given the start state s and the action a. Since the actions of MDPs are not predicable but observable, one way to obtain and maintain the probabilities in the transition table is simply by counting the occurrences of the resulted state.

Under the restricted settings, a *passive RL* is defined as the learning of the utilities given a fixed policy. It is to perform the policy evaluation step in the policy iteration algorithm when there are updates in the transition table. Comparing with the passive learning, an *active RL* not only evaluates the actions which are already performed, but also explores actions which have not been tried before. In particular, it trades off exploitation against exploration.

RL task with a MDP model can be regarded as the learning of the optimal policy (π^*).

Taking the curiosity into consideration, the Bellman Equation 4.1 is updated and shown in Equation 4.2. $f(u, n)$ is a exploration function in the equation, where u is the utility of state-action pair, n is the number of the occurrences of this state-action pair. Equation 4.3 shows a simple definition of exploration function. When the recorded occurrence number of a state-action pair is smaller than a predefined value N_e, R^+ is taken as its utility. In the exploration function, N_e is the threshold for a well-learned action. R^+ is the rewards for trying an unknown action.

$$U^+(s) \leftarrow R(s) + \gamma \max_a f(\sum_{s'}(T(s,a,s')U^+(s')), N(s,a)) \tag{4.2}$$

$$f(u,n) = \begin{cases} R^+, \text{if } n < N_e \\ u, \text{otherwise} \end{cases} \tag{4.3}$$

If Equations 4.2 and 4.3 are taken in the policy iteration algorithm, the RL for a MDP model with finite search space is to perform the updated policy iteration algorithm whenever there is an update in the transition table. This algorithm is called *Adaptive Dynamic Programming*, as shown in Figure 4.4

.

- REPEAT
  - Update transition table by counting the occurrences of the state action pairs.
  - Run the policy iteration algorithm using Equation 4.2
- UNTIL there is no update in the transition table and the optimal policy is converged.

Figure 4.4: Adaptive Dynamic Programming with a finite search space MDP model

This thesis implements an adaptive dynamic programming algorithm for the action selection of StarKick.

4.3 Dyna Architecture

A class of architectures for intelligent systems based on approximating dynamic programming methods are given by Richard S. Sutton in [7]. Dyna is one of the integrated architectures for learning, planning, and reacting. It may contain the following parts. [8]

- Trial-and-error learning of an optimal reactive policy, a mapping from situations to actions.
- Learning of domain knowledge in the form of an action model, a black box that takes the input as a situation and action and outputs a prediction of the immediate next situation.
- Planning: finding the optimal reactive policy given domain knowledge (the action model)
- Reactive execution: No planning intervenes between perceiving a situation and responding to it.

Three major components of Dyna architecture is depicted in [8]. They are given as follows.

- The structure of the action model and its learning algorithms;
- An algorithm for selecting hypothetical states and actions.
- A reinforcement learning method, including a learning-from-example algorithm and a way of generating variety in behaviors.

A generic Dyna algorithm is given in Figure 4.5.

REPEAT FOREVER
Observe the world's state and reactively choose an action based on it;
Observe resultant reward and new state;
Apply reinforcement learning to this experience;
Update action model based on this experience;
Repeat k times:

1. Choose a hypothetical world state and action;
2. Predict resultant reward and new state using action model;
3. Apply reinforcement learning to this hypothetical experience.

Figure 4.5: Generic Dyna algorithm [8]

From the descriptions above, we can find that the Dyna architecture is surprisingly suitable for this thesis. Trial-and-error learning and action model can be easily implemented according to the observations during a table soccer game. MDPs provide a structure for

the planning and reinforcement learning. The exploitation of RL implements learn-from-example and the exploration will generate varieties in behavior. In order to support the reactive execution, the policy of MDPs can be used for the execution even if it is not converged. The solving of MDPs could be run in the way that it never interfere with perceiving a situation and responding to it.

As we have introduced in Chapter 2.2, there are two different previous approaches for the action selection of KiRo. The decision tree approach works reactively, which uses the architecture of a reactive system. The decision-theoretic planning approach constructs a planning tree for the reactive executions, which uses the architecture of a conventional planning. The architectures of the reactive systems and the conventional planning are shown in Figure 4.6 (a) and (b). Comparing with the previous approaches, Figure 4.6 (c) shows the architecture of Dyna (incremental dynamic programming), which is going to be realized in this thesis.

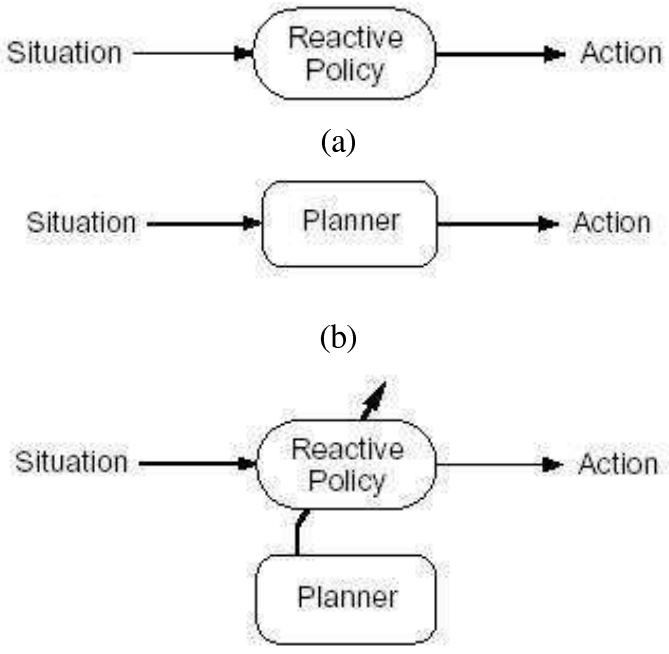


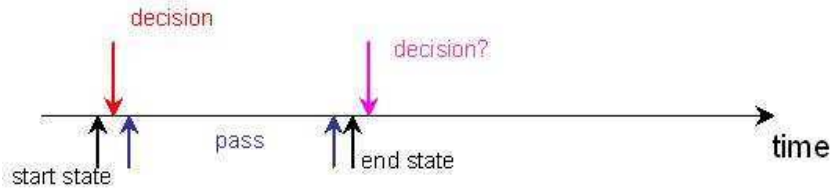
Figure 4.6: Simplistic comparison of the architectures: (a) reactive systems, (b) conventional planning, and (c) Dyna architecture [8]

Chapter 5

Action Selection of StarKick

A naive way of solving the action selection problems of StarKick using MDPs is to create the states by using the grids on the playing surface and considering the moving speed as well as the direction of the ball. If we roughly create 8 moving directions, 2 moving speeds of the ball, and taking the grids of $50mm \times 28mm$, there is going to be 9216 states created in the MDP model. In Chapter 3 we have developed 18 different actions, so that a MDP model with the transitions of state-action-state triples would have 1528823808 transitions. Of course, these transitions can be significantly reduced if we don't consider the impossible situations. For example, we can ignore a transition if there are 1000mm between its start state and end state. However this kind of reductions seems far from enough. Then the problem arises on how to use the domain knowledge to reduce the transitions of the MDP model.

The MDP model in this thesis employs a small state space. The actions in the model are complex, so that the transition number of the MDP model is significantly reduced, and the final program, which implements the MDP and RL algorithms, can be run under the current computation conditions to support the reactive execution in a real table soccer game. Figure5.1(a) shows the situation of using one of the *PassActions* in the MDP model. Imaging that the ball is moving with a low speed within the control range of a rod after the passing, we can find that in order to make a decision in this situation, there should be a state consisting the position, moving speed, and direction of the ball. It's similar to the planning in the decision-theoretic planning approach. In this thesis, more complex *DribbleActions* are developed, so that we avoid considering a state with the moving direction of the ball. Figure5.1(b) shows the same situation but using complex *DribbleActions* in the planning. In the figure, the *DribbleActions* are much longer; it continues until MDP can make a further decision to "kick" or "dribble"again at the "end state" This way significantly reduces the number of the MDP transitions, and makes the planning happen exactly when there is a chance to select an action from the applicable action set.



(a)

(b)

Figure 5.1: (a)The *PassActions* in the planning and (b) the complex *DribbleActions* in the planning

This approach is different from the previous approach that uses the decision-theoretic planning. The approach here has a finite search space. The sequence of actions depends on the observations, which is a, so called, closed-loop planning. The decision-theoretic planning approach employs forward simulation and heuristic function, where the planning is performed in the absence of observations, and a so called open-loop planning [2].

The first part of this chapter models the action selection of StarKick using MDPs, in which four modules are created. The second part of this chapter introduces two opponent models, which assume two different sets of opponent behaviors. And the third part explains how the transitions of MDPs are learned by RL.

5.1 The MDP Model

This section models the action selection of StarKick using MDPs with limited search space. MDPs are presented as states, actions, rewards, and transitions, shown in Figure 4.1. There are 146 states, 20 actions, and therefore 426320 transitions if the transitions are the triples of state-action-state which are created by the potential of these sets. Four modules of MDPs are created which significantly reduce the potential transitions in this section.

5.1.1 States

The actions described are used as a clue to create the states. The purpose of MDPs is to find the optimal policy, which can be regarded as choosing the best action from the applicable action set. All possible actions implemented on StarKick were already given in Chapter 3. Some of them are unique, such as *StopActions* and *Clear*, which are only applicable in a particular situation, in which there are no other choices. Some actions share the same preconditions, such as *KickActions* and *PassActions*. These preconditions are used as a clue to create the MDP states, so that the optimal policy of a state becomes the best choice in a particular action set.

The states of MDPs are defined as the union of *KickStartStates*, *DribbleStartStates*, *OpponentStartStates*, and *GoalStates*, as shown in Figure 5.2. Four sets are used here which makes the presentation of four modules much easier. The *PassActions* can be performed in the *DribbleStartStates*. It is called *DribbleStartStates* because *PassActions* are combined to more complex *DribbleActions* later.

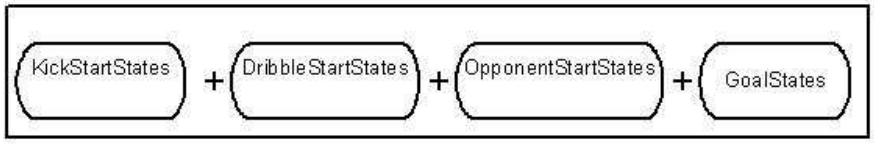


Figure 5.2: State sets of the MDPs.

KickStartStates comes from the precondition of *KickActions*, which assumes that the ball is still or moving very slow in the kick-able band. The ball's position within a particular kick-able band is classified into five different classes for defender, midfield and attacker. They are Left1, Left0, Middle, Right0, and Right1. For goalkeeper there are only three classes: Left0, Middle, and Right0. These 18 regions are shown in Figure 5.3.

Every region has the similar width around $136mm$ because that is how the basic actions *PassParallel* moves the ball. No other actions change the position of the ball smaller than *PassParallel*, so that the accuracy of the states matches the accuracy of the actions. The state set here is the so called *KickStartStates*. Considering that there are 18 different regions and the ball could be still or moving slowly, 36 states are defined in this state set.

DribbleStartStates comes from the precondition of *PassActions*, which assumes that

Figure 5.3: Kick-able regions

the ball is still and under control. When the ball is still, there are two different situations for the rod, which depends on whether the figure locks the ball or not. In addition, the ball is not forward kick-able in the "negative lock band", not backward kick-able in the "positive lock band", and when it is still in the middle "kick-able band" only touch actions are available to pass it parallel. Besides the five classes mentioned in *KickStartStates*, the *DribbleStartStates* need to consider both the rod and the bands. Figure 5.4 shows all these considerations of the states. In the figure, "Positive Lock Band" is shown in blue. "Negative Lock Band" is in red, and "Kick-able Band" is in yellow. Right and Left regions are bounded by the black lines. Please notice that not all of regions are shown. There are all together 90 states defined in *DribbleStartStates*.

The *OpponentStartStates* assumes the preconditions that the ball is controlled by the opponent. Actually there are no other choices than performing *SoftBlock* on all StarKick's rods when the opponent controls the ball. It seems that MDP mechanism is not necessary because there are no choices to choose from. However, loosing the control is a very possible end for all *KickActions* and *PassActions*. Ending up with a situation that the opponent's attacker is controlling the ball should be much worse than ending up with the situation that the opponent's goalkeeper is controlling the ball. Extra MDP states should therefore be created to evaluate what the situations are, although there is no decision to be made.

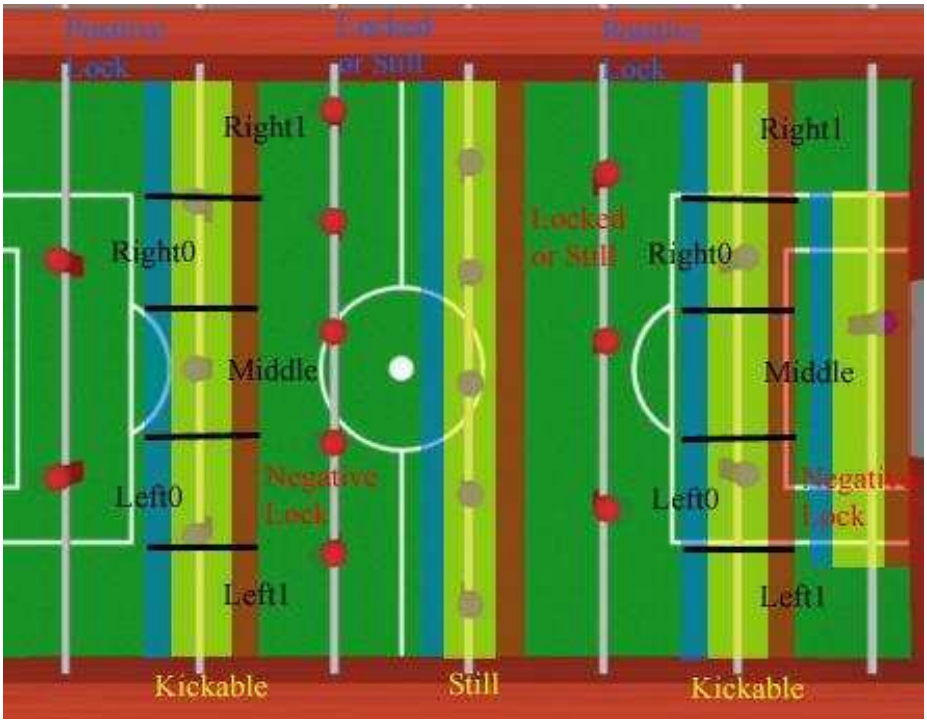
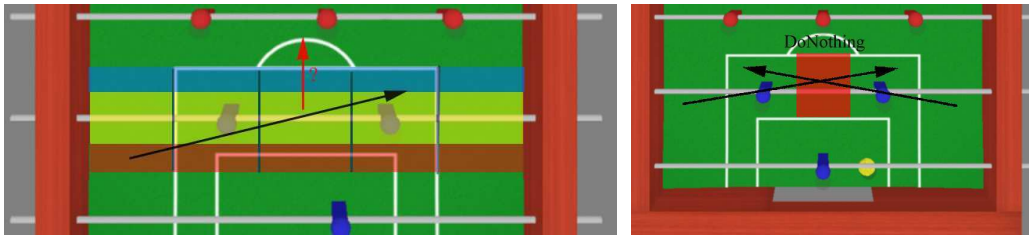


Figure 5.4: *DribbleStartStates*

It is very easy for a skillful human being to pass the ball from the defender to the goalkeeper or vice versa, and actually the actions are the same regardless of whether the opponent's defender is in control of the ball or the opponent's goalkeeper is in control of the ball. For this reason, it is not necessary to tell whether the ball is controlled by the opponent's defender or goalkeeper. For the same reason, when the opponent's defender is in control of the ball, identifying where exactly the ball is, within the opponent's defender controlling band, does not make sense. Then there are three different states here: *OpponentDefender*, *OpponentMidfield*, and *OpponentAttacker*. These states are called *OpponentStartStates*.

An alternative to evaluate these situations is to create three end states for the different rods of the opponent, or using the heuristic function presented in decision-theoretic planning [9]. Considering that the game is still going on, and the end states in MDPs should be a negative target or positive target where all planning is ending up, *OpponentStartStates* is regarded as a set of common MDPs' states, whose utilities are decided by the transitions started with them.

Two apparent end states are that the ball is in the opponent's goal (*OpponentGoal*) and the ball is in the own goal (*OwnGoal*). These states are regarded as the final targets of



(a) (b)

Figure 5.5: Take the basic actions in the *Dribble* module

MDPs, and called *GoalStates*.

There are some "transient states" in which the ball moves in very high speed and maybe bounces several times among the figures and the walls, both opponent and StarKick are trying to slow it down and control it. The only available action of StarKick in this situation is *SoftBlock*. The "transient states" are defined as those states, in which the reactions are given fixed, and it is not necessary to evaluate how good these situations are. The "transient states" actually never enter the MDPs, while working constantly in the action selection model.

5.1.2 Actions

Taking the states described in Section 5.1.1, the *PassActions* in Chapter 3.3 can not directly be used by the action selection of StarKick. Figure 5.5 (a) shows an example of directly taking these actions. At the beginning, the ball is still at *Left1DefenderNegativeLock*. Assuming it is passed by action *PassLongBackRight*, it goes along the arrow and arrives at the state *LowSpeedRight0Defender* finally. Please notice that along the moving track it actually reaches two more states before the final one. These states are *LowSpeedLeft0Defender* and *LowSpeedMiddleDefender* one after another. Obviously the probabilities for the ball reaching "Right0" would be much lower if one action in KickActions was performed at these previous states. That is, the transition from one state to another not only depends on this state, but also depends on the policies of other states, which conflicts with *Markovian assumption* introduced in Section 4.1.

There are several possible solutions. One solution would be combining the kick actions along the trial with the start state of the passing to create a more precise state. Clearly, this way makes the state space significantly big. Another solution would be creating an extra *DoNothing* action applicable on the states along the trial. However, the states of MDPs in Chapter 5.1.1 did not include the moving direction of the ball, which made a *DoNothing*

transition ambiguous. For example the two pass actions shown in Figure 5.5 (b) both have a state *LowSpeedMiddleDefender* along the passing trial, where *DoNothing* action should be employed. The resulted states for the two passing actions should be unified to transitions started with state action pair *LowSpeedMiddleDefender* and *DoNothing*, which is not powerful enough to describe these two passing actions.

Tow alternative solutions on this problem are implemented and compared in this thesis, both of them create bigger actions by combining one or more basic actions. The first one is called *Combined-Dribble*, the second one is called *Single-Dribble*

The *Combined-Dribble* approach has combined dribble actions expressed as *(Start, Target)* where Start is from *DribbleStartStates* and Target is from *KickStartStates*. The available actions at the state *Lef1DefenderNegativeLock* are shown in Figure 5.6 in black arrows. Please notice that these actions are UNIT actions. That is, although they may be implemented by performing several basic actions one after another, there shouldn't be a "break" in the action sequence. All these actions are called *DribbleActions*.

Figure 5.6: The applicable *Combined-Dribble* actions in one state

The *Single-Dribble* approach has dribble actions expressed as *(Action, Target)*, where Action is one of the *PassActions* described in Chapter 3.3, and Target is from *KickStartStates* as the first solution. The preformance of the *Single-Dribble* action in this solution is as follows. At the beginning the ball is passed by one of *PassActions* described by Action in *(Action, Target)*. When the ball is rolling after the passing, there are two situations. The first situation is that the ball is in Target. Dribble action already finishes its task at this situation, and the action selection goes on from a state in the *KickStartStates*. The second situation is that the ball is not in Target. The *Single-Dribble* action goes on by performing LockSlow to actively stop the ball, which results in a till ball, a locked ball, or a lost ball. The action selection decides to defend in the case of loosing control or dribble again in the case of having a still or locked ball. Figure 5.7 shows the activity diagram of dribble

actions in the second solution.

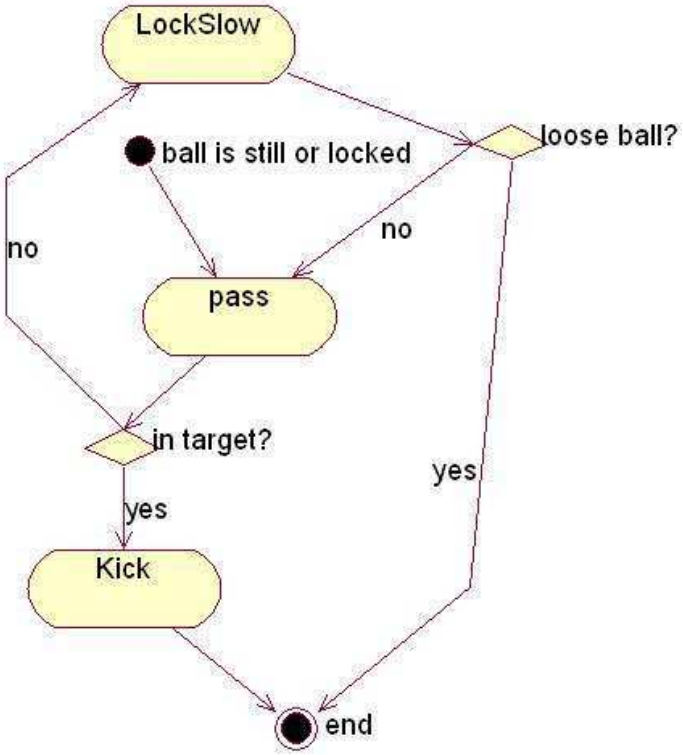


Figure 5.7: Activity diagram of the dribble action *(Action, Target)*

A possible extension of the second solution is to have more targets, which can be expressed as *(Action, TargetList)*. Figure 5.8 shows one possible action in state *Left0DefenderNegativeLock*, which is expressed as *(PassLongBack, [MiddleMoving, Right1Moving])*. Obviously, the number of actions in this extension is significantly bigger than the second solution.

After defining the combined dribble actions, the second element of MDPs, action set A 4.1, can be defined as *DribbleActions* together with three action-sets which are *KickActions*, *Clear*, and *StopActions* defined in the chapter 3, as shown in Figure 5.9.

5.1.3 Four modules of the MDP model and the transitions

The actions implemented on StarKick are structured to four sets, in which only *KickActions* and *DribbleActions* have more than one choices. Since *StopActions* and *Clear* are unique to some situations, the action selection of StarKick is to make the following two decisions:

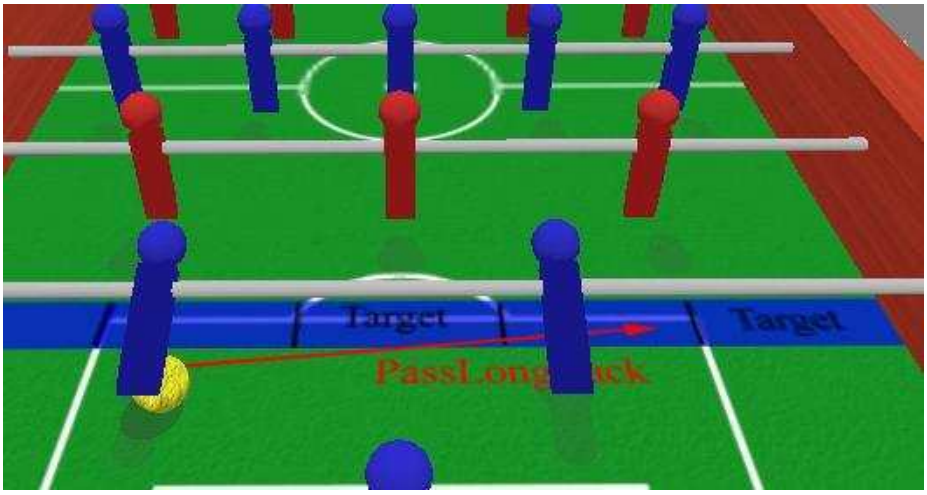


Figure 5.8: An example of the *(Action, TargetList)*

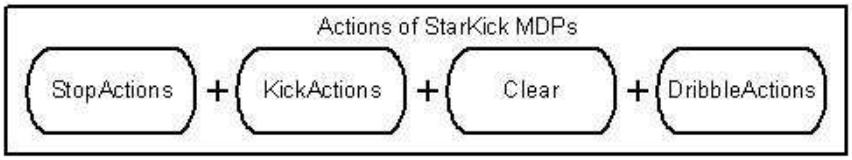


Figure 5.9: Action sets of the MDP model

choosing an action from *KickActions* when the ball is kick-able, and choosing an action from *DribbleActions* when the ball is still and under the control.

The "playing" process of a rod is defined as the following. Firstly the rod gets the control of the ball. Then the ball is kicked back immediately, which is so called *KickImmediately*, or it is stopped. After that, the ball will be dribbled for a while, in case it was not kicked back before. Finally the ball is kicked, and the rod looses its control, which is called *DribbleKick*. There are two chances *KickImmediately*, or *DribbleKick* to kick the ball.

KickImmediately is different in MDPs from *DribbleKick* because human beings react differently to a *KickImmediately* action and a *DribbleKick* action in the real game. Although the two kick process in the MDP share the same start states, actions, and end states, their results could be totally different. In other words, the probabilities for a successful kick action taking place after dribble would be another story when the ball is kicked back immediately without any dribble. In order to make the transitions more realistic, a *KickIm-*

mediatelyStartStates is added into the MDP to model the kick immediately process. They are the states when the ball is rolling slowly in the 18 regions of Figure 5.3.

Modules of MDPs are induced due to the structured states and actions in the playing process, where the subsets of states and actions work together when they are applicable. Those transitions, which are never applicable, can be cut off. If we take *(Start, Target)* as the dribble actions, the number of transitions is significantly reduced by these modules. Since the state and action sets of different stages are distinct from one another in the process, four modules of MDPs are designed to model them separately. They are *Opponent* module, *KickImmediately* module, *Dribble* module, and *DribbleKick* module. The end states of one module could be the start states of another. The *Opponent* module is not for choosing an optimal action, but for estimating how bad it is if the opponent gets the ball.

Two alternative dribble modules are implemented, they are *Combined-Dribble* module and *Single-Dribble* module, taking *(Start, Target)* or *(Action, Target)* as dribble actions separately. Figure 5.10 (a) shows four MDP modules, in which *(Start, Target)* is taken as dribble actions, and the transitions are reduced to 31005. Figure 5.10 (b) shows the *KickImmediately*, *DribbleKick* and *Dribble* modules, in which *(Action, Target)* is taken as dribble actions, and there are 229905 transitions.

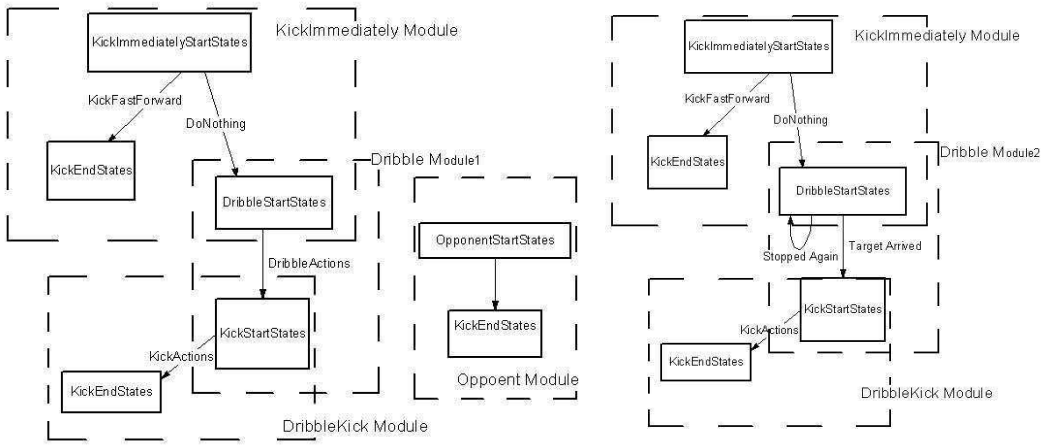


Figure 5.10: MDP modules in the "playing" process (a) *(Start Target)* as the *DribbleActions* and (b) *(Action, Target)* as the *DribbleActions*

In order to study these modules, an extra state set, *KickEndStates* has to be introduced here. The *Opponent* module start when one of the opponent rods controls the ball, and end when the rod looses the ball. The start states are the *OpponentStartStates* which was already defined in Chapter 5.1.1. The end states of the *Opponent* module are called *Kick-*

EndStates, which is often indicated by the situation that the ball has a controllable low speed or is still. The *KickEndStates* should include all these controllable or still situations, which are exactly the union of the *KickImmediatelyStartStates*, *DribbleStartStates*, *OpponentStartStates*, and *GoalStates*. The elements in *KickEndStates* are shown in figure 5.11. It is also the end state set for the *KickImmediately* module and the *DribbleKick* module. The *KickStartStates* is the start state set of the *DribbleKick* module, which was given in Chapter 5.1.1.

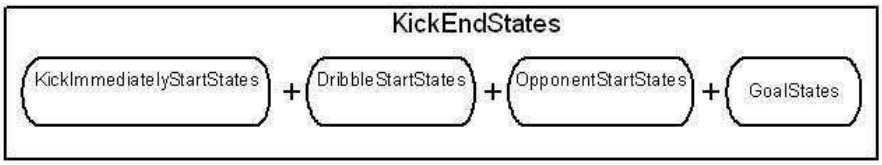


Figure 5.11: *KickEndStates*

There are four applicable action sets for four different modules of the MDP. The action set for the *Dribble* module is the *DribbleActions* defined in chapter 5.1.2. The *Opponent* module does not care for the actions. All rods perform *SoftBlock* when the opponent controls the ball. Only *KickFastForward* is the available kick action in the *KickImmediately* module, because the speed of kicking is emphasized. Another available action other than kick in the *KickImmediately* module is *DoNothing*, the ball will keep on rolling in low speed until it is stopped by the *StopActions* which work in a reactive way and is not in any modules. The *DribbleKick* has choices of all actions in the *KickActions*.

Taking *(Start, Target)* as dribble actions, the end states of the *Dribble* module are tricky because the actions of dribble are associated with the Target. If the Target is regarded as a part of action, the end states of the *Dribble* module become *Success*, *NegativeLoose*, and *PositiveLoose*. In order to evaluate the utilities of every state-action pair, the *Success* is mapped to Target in the action, the *NegativeLoose* is mapped to the next rod in negative side, and the *PositiveLoose* is mapped to the next rod in positive side. For example, in the state *DefenderRight1NegativeLock*, if the action "pass to *LowSpeedLeft0*" is taken, ending up with *Success* means that transition ends up with state *LowSpeedDefenderLeft0*, which is a start state of the *DribbleKick* module; ending up with the *PositiveLoose* means that the end state is *OpponentAttacker*; ending up with the *NegativeLoose* means that the end state is *Goalkeeper*. The transitions in the example are shown in figure 5.12.

Taking *(Action, Target)* as the dribble actions, the end states of the *Dribble* module are the *Success*, *NegativeLoose*, *PositiveLoose*, and those states in *DribbleStartStates* which

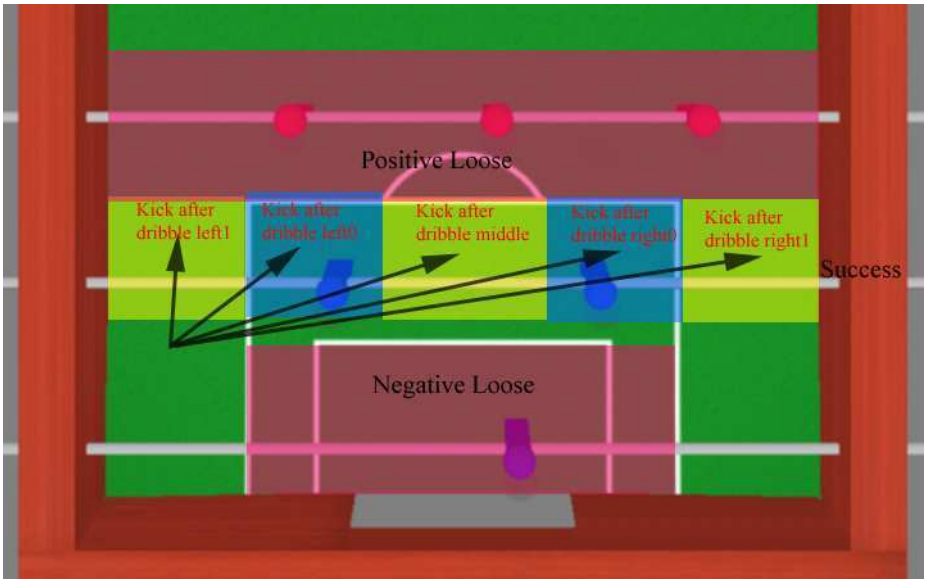


Figure 5.12: The transitions of the *(Start, Target)* dribble

are related to the current rod. Figure 5.13 shows the end states of the *Single-Dribble* module at state *Left1DefenderNegativeLock*.

5.1.4 Rewards

In table soccer game the rewards come from the score, therefore the states *OwnGoal*, and *OpponentGoal* have the ultimate rewards -1 and 1. In order to prevent the ball from staying in one state, other states should have small negative immediate rewards. It should be small enough so that StarKick will not kick the ball into *OwnGoal* to commit suicide. Here -0.001 is used for all other states' immediate rewards, that is, when StarKick has to perform more than 1000 actions to achieve *OpponentGoal*, StarKick will prefer to commit suicide.

These rewards could be adjusted according the strategy. Increasing the rewards at opponent's goal would make StarKick to be more aggressive, while being scored more easily. Decreasing the rewards for the states other than *GoalStates* will give those actions, which are safer but less effective, more chances.

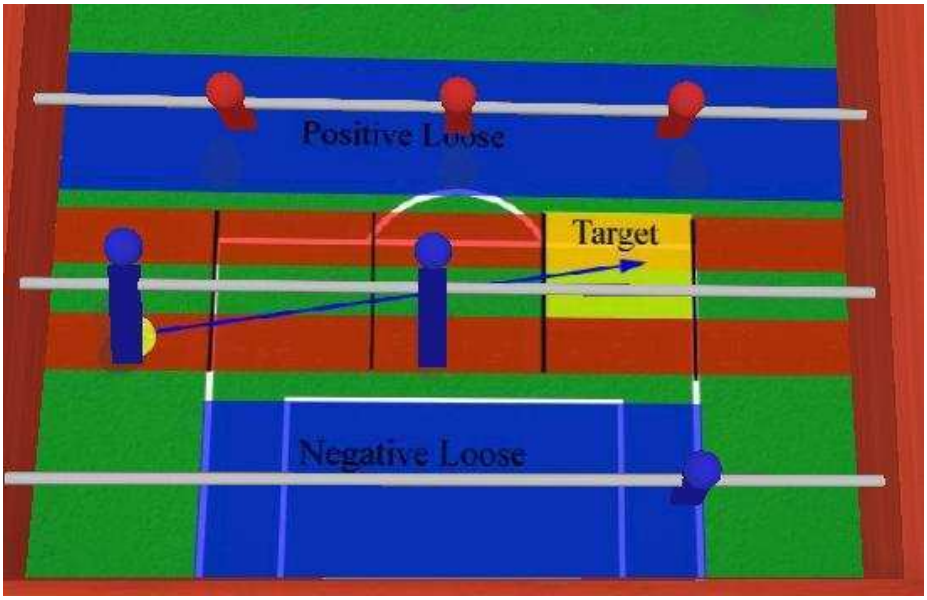


Figure 5.13: The *DribbleEndStates* of the *Single-Dribble* module.

5.2 Opponent Models

Two opponent models are created here, each assuming some particular opponent behaviors. They are *Blocked* model and *Dynamic* model.

The opponent takes the role of the teacher in the *Blocked* model. Only one opponent plays with StarKick without significant changes of the playing skills. The model is called "blocked" because the opponent would always be trying to block the ball when StarKick is controlling it. StarKick learns the general evaluation for different kicking actions in this model, which is a basis for the decision in *Dynamic* model.

The opponent would be intelligent and different human beings in the *Dynamic* model. "Intelligent" means that the opponent would be familiar with the policy if it is repeated again and again by StarKick; and therefore significantly prevent it from being successful. For example, if *KickFastForward* is an optimal policy at the state *MiddleAttacker*, and StarKick repeats it for four times, the fifth time would be hardly to success because the opponent would expect it when the ball is at the place. "Different" means that the opponent is changing all the time, each of them may play only one game, and then a stranger with very different skills comes. The policy which was good for the first player may become very bad for the second.

5.3 Implementation of the Reinforcement Learning

The RL algorithm works in the following way. All transitions are recorded in the game by counting; they are saved before the StarKick is turned off, and load again when StarKick is started up next time. Whenever there is an update in transition table, the policy iteration algorithm should be run until all utilities are converged. The following sections discuss the RL algorithms against the *Block* and *Dynamic* models separately.

5.3.1 Learning from the *Blocked* Model

Learning from the *Blocked* model in StarKick is to give MDPs the strategy on how to deal with transition probabilities before they can be accurately evaluated by the examples from the real games. At the beginning of the learning, all recorded transition numbers are set to zero. The learning program updates these records during the games. Before policy iteration, the recorded transition numbers should be normalized. A mechanism which makes StarKick curiously explore the unknown action probabilities is implemented in the normalization program, which was already described in Equation 4.3. For the *Dribble* module, when the recorded number of a particular state action pair is less than ten, the normalization program makes the dribble transitions always (100%) end up with the *Success*. For the *KickImmediately* and *DribbleKick* modules, kick transitions are set to very good results such as *OpponentGoal*; similarly, For *Opponent* module the ball is assumed to pass to the next opponent rod, and the opponent's attacker can always shoot the ball into the *OwnGoal* (100%). The threshold of the record number of kick transition is set to five. The threshold of record number of opponent's transition is set to ten.

5.3.2 Learning from Dynamic Model

Problems would arise when the opponent is changed constantly in the *Dynamic* model, because the current opponent would always be a stranger to StarKick. If the learning scenario in the *Block* model is used in the *Dynamic* model, the more transitions are learned, the blunter KiRo would be to a stranger. For instance, a *KickFastForward* is supported by 1000 records before, although it is a stupid action for the new comer, it will still be repeated again and again by StarKick as optimal policy until its utilities become lower than the second best action, which would be much slower than loosing the game.

The solution here is to give the new features of the strangers more weight. In addition to the *Blocked* model, the RL algorithm here gives different "weights" to the transitions which happened recently, so that StarKick has a way to avoid repeating the mistaken, while trying other good choices with some curiosity. The transitions which happened recently are

recorded in a list, sorted by time. In practice, the RL module of StarKick takes the following configuration. In the *Opponent* module, the recent ten transitions of each start states take 30% of the total probabilities, so that StarKick will not risk losing the control if the opponent is very skillful. In the *KickImmediately* and *DribbleKick* modules, the most recent transition takes 15% of total probabilities, the second takes 10%, and the third takes 5%. By this way, any start states of kick tries the best (optimal) action at the very first time, a failure will result in second best action (in the case 15% makes sense), and so on. The *KickImmediately* and *DribbleKick* modules of the MDP model have the curiosity of trying a maximum of four kick actions on a stranger, ordered by their utilities learned in the *Blocked* model. Any successful result will cause the same action to be selected again. This way does not abuse the "curiosity", but tries the second and maybe the third better action indicated by the *Blocked* model. Figure 5.14 shows how the "kick" transition records are normalized in the *Dynamic* model.

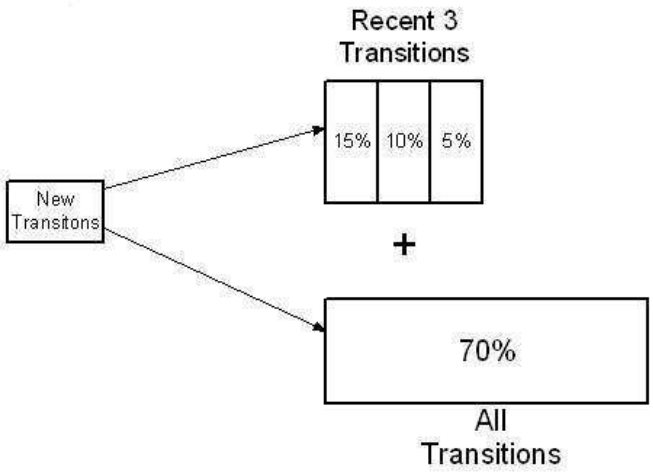


Figure 5.14: Normalizing the *Kick* transitions in the *Dynamic* model

Chapter 6

Software Implementation

The action control and action selection model described before are implemented in the KiRo program, which was developed for KiRo and adapted to StarKick. This chapter describes the problems and solutions when it comes to the implementations.

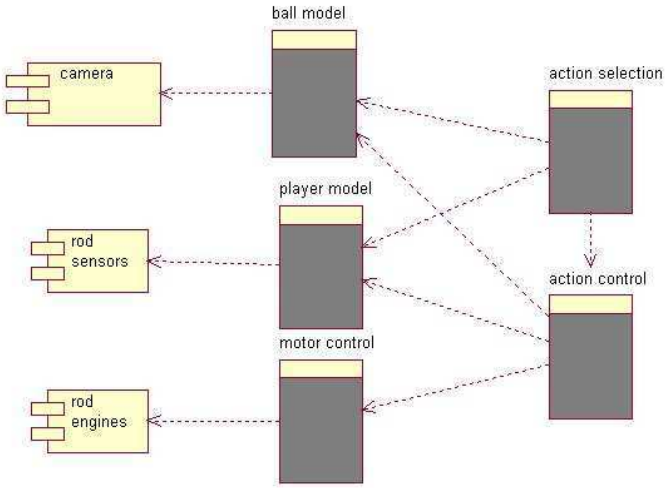
6.1 Environments

KiRo program was developed on a SuSE Linux platform. The programming language is C++. QT is used as a graphic user interface library. This thesis is based on the KiRo program which already has the software drivers for the hard-wares, the architecture for the action control and action selection, vision system, and world model including ball model and player model.

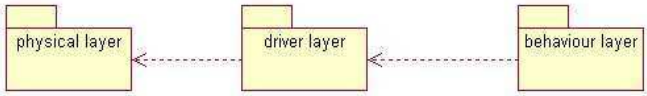
The software environments of this thesis are described as the component view shown in Figure 6.1 in UML manner. The KiRo program are divided into three layers. The physical layer contains hardware components such as sensors, camera, and engines. The behavior layer contains the action selection and action control model, which are the main part of this work. The layer between them is called the driver layer, which can be seen as a friendly interface for the behavior layer to manipulate the hardware. The arrow lines in the figure are used to depict the dependencies. Figure 6.1 (b) shows the dependencies among these layers.

6.2 Implementation of Action Control

A simple action becomes difficult when it is implemented in a dynamic noisy robotic system. This section describes the difficulties and solutions in developing basic actions, and



(a)



(b)

Figure 6.1: (a) Component of KiRo Behavior System. (b) Layers.

action games.

6.2.1 Difficulties and Solutions

To hand code the very basic actions of StarKick, there are some difficulties which need to be mentioned here.

The first difficulty is that no action can be finished in one cycle. As mentioned in Chapter 2, the processing cycle should be $20ms$. However even a very simple action, for example raising the rod from $0°$ to $90°$, takes more than this limitation. In order to overcome the difficulty, the action control sends feedbacks to the action selection, informing that one action is done or not. The action selection decides whether or not to break the performing of one action. For example, the *LockFast* action tries to lock the ball in whatever situation unless the ball is locked or action selection tells the rod another action.

The second difficulty is that all actions depend on time. The speed and position of the ball are very important to all actions. For example, in order to lock the ball, the *LockFast* must consider the moving speed of the ball, and the turning speed of the rod. Then it makes

the figure press the ball when the ball is in the lock band. In order to overcome this difficulty, the action control of StarKick predicts the ball's position by the moving vector of the ball, and performs the action by predefined time parameters. For example, in "Lock Ball" action, the speed of the ball in the x direction is used to calculate the exact point of time that the ball reaches the "Lock Band", and before a predefined time amount, the rod begins to turn to achieve this action.

The third difficulty is that the hand-coded actions are hard to debug. Since the action of the rod is highly dynamic, it is hard to reproduce any actions. In practice, this difficulty is overcome by using video recorder to record the playing, and locating the bugs in the program by replaying the records and checking the print out.

6.2.2 Action Games

In order to test basic hand-coded actions, the following small games are created, in which a particular set of actions are repeated again and again, so that the predefined parameters can be adjusted and the performance of the actions can be improved during these games.

The first game is for the *LockFast*. One of the opponent's rods passes the ball to StarKick with medium speed. The rod, which the ball is rolling to, performs a *LockFast* action. After the ball is locked, StarKick stops reacting to the ball until it is put under an opponent rod again. Then the pass and *LockFast* are repeated.

The second game is for the *PassLongBack*. The beginning of this game is the same as the first game. After the ball is locked, the rod performs the *PassLongBackRight* if the ball is at the left side of the rod, and performs *PassLongBackLeft* if the ball is at the right. After passing, the ball is locked again by the *LockSlow*. The passing and locking are repeated until StarKick looses the control. The *LockFast* is performed again in this situation and the game goes to the beginning.

The third game is for the *PassShortBack*. The beginning of this game is the same as the first game. After the ball is locked, the rod performs the *PassShortBackRight* if the ball is at the left side of the rod, and performs the *PassShortBackLeft* if the ball is at the right. After the passing, the ball is locked again by the *LockSlow*. The passing and locking are repeated until StarKick looses the control. The *LockFast* is performed again in this situation and the game goes to the beginning.

The fourth game is for the *PassParallel*. The beginning is the same as the first game. After the ball is locked, the rod performs the *PassParallelRight* if the y coordinate of the ball is smaller than a predefined value, and performs the *PassParallelLeft* if the y coordi-

nate of the ball is smaller than another predefined value. These two predefined value are near the two side walls, so that the ball is passed parallel in one direction until it reaches the end. After passing, the ball is locked again by the *LockSlow*. Then the passing and locking are repeated as before.

The fifth game is for the *Touch*. This time the ball is stopped by the *SoftBlock*. After the ball is still, the playing figure touches the ball to move it to the right if the ball is in the left side of the playing figure, and to the left if the ball is in the right side of the playing figure. *LockSlow* is used to stop the ball, but the game comes to an end after the ball is locked, and can be started again if a human being removes the locked ball and passes it again from one of the opponent's rods.

The sixth game is for the *KickImmediately*. The ball is stopped by the *SoftBlock*. As soon as the ball is kick-able, it is kicked by the action *KickFastForward*. Then it stops reacting until the ball is under control of the opponent. The *SoftBlock* and *KickFastForward* can be repeated.

The seventh game is for the *KickSlowLeft* and *KickSlowRight*. The ball should be passed from the negative direction. StarKick performs the *LockFast* to lock the ball. When the ball is locked and still, the *KickSlowRight* is used to kick it if the ball is in the left side of the rod, and the *KickSlowLeft* is used to kick it if the ball is in the right side of the rod. After the kicking, the rod stop reacting to the ball until it is under control of the opponent. The *LockFast*, *KickSlowLeft* and *KickSlowRight* can be repeated.

The eighth game is for the *KickFastRight* and *KickFastLeft*. In contrast to the kick actions in the seventh game, the *KickFastRight* and *KickFastLeft* are used as kick actions in this game, while other settings are the same as in the seventh game.

6.3 Implementation of Action Selection

Action selection model using Markov Decision Processes with Reinforcement Learning is the final goal of this thesis. In practice, this final goal is however difficult to be attacked directly, because every tiny bugs would mess up the whole behaviors. It is even harder than the difficulties we met in the implementation of the basic actions because action selection model is more complex than a simple action. this thesis employed two more opponent models to implement the action selection model step by step. Every step achieves a subgoal which limits the problems, and finally solved the action selection problem.

6.3.1 Control Program for the MDP model

In the action selection model, a control program is constructed, which can tell the start and the end of a particular module of MDPs according to the current state, and makes the decisions by the MDPs. Figure 6.2 shows the sequence diagram of an example on how the control program works. In the example, firstly the ball is under the control of the *OpponentDefender*; it is passed to the *OpponentMidfield*; the *OpponentMidfield* looses it by a control failure; StarKick decides to the *KickImmediately* on its midfield, then the attacker of StarKick gets the control, it dribbles a little bit and kicks; finally game ends in the *OpponentGoal*.

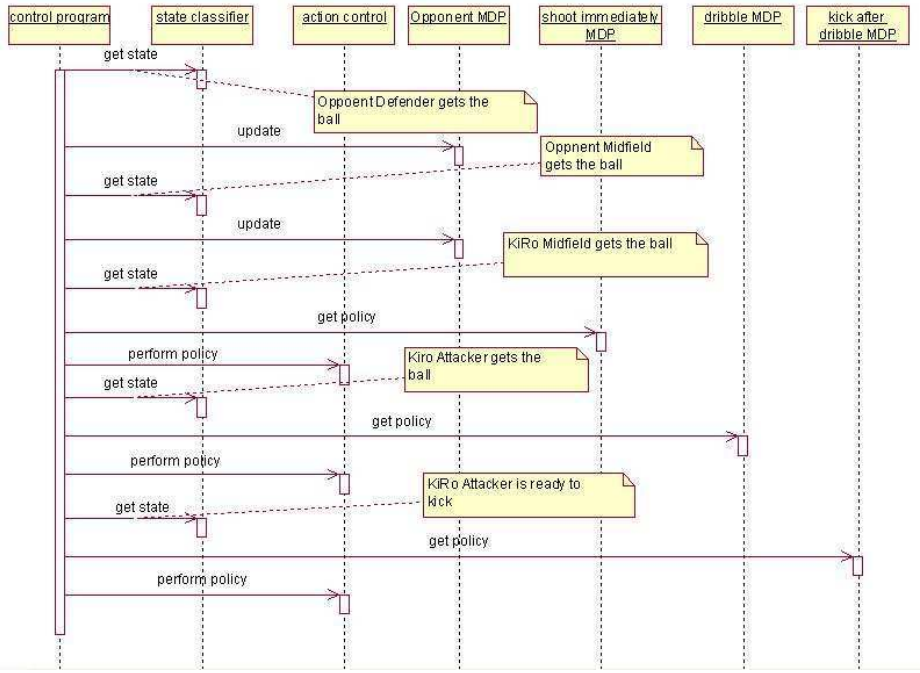


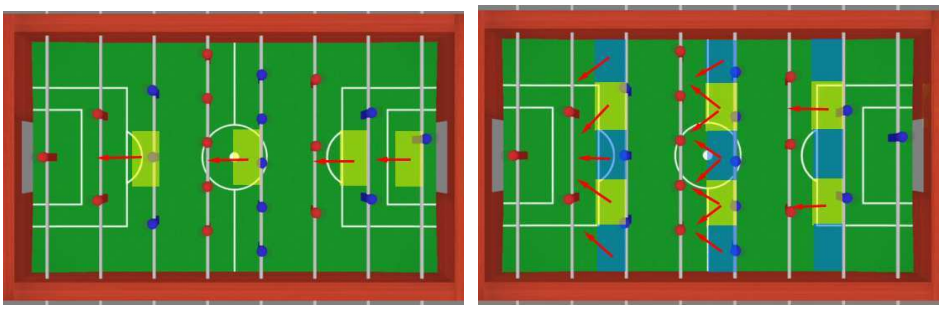
Figure 6.2: The "control" program

6.3.2 Extra Opponent Models for Implementation

Motivated by simplifying the problems, two more extra opponent models are added to the system in the implementation, resulting in four opponent models all together. From simple to complex, they are *Clear* model, *Centered* model, *Blocked* model, and *Dynamic* model.

The *Clear* model assumes that there is no opponent at all. It is implemented by making all opponent's playing figures upside down. Whenever a rod gets the ball, it tries to pass the ball to the middle, and shoot the ball with the action *KickFastForward*. The kick pattern for

the *Clear* model is shown in Figure 6.3.2 (a). The kick regions are in yellow. When the ball is still or moving with low speed in these regions, the rod performs the *KickFastForward*. When one rod is kicking, other rods in front of the ball perform the *Clear* action.



(a) (b)

Figure 6.3: The kick patterns of the (a) *Clear* model and (b) *Centered* model

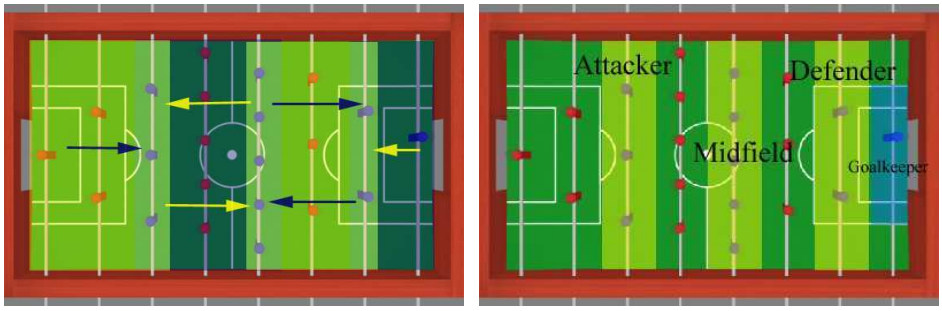
The *Centered* model assumes all the opponent's playing figures are still and centered. The opponent has no hostile intention, whenever he gets the ball, the ball will be passed back to StarKick with friendly medium speed. In addition, StarKick knows exactly where the opponent rods are, it tries to kick the ball with slow speed at the position between the two opponent figures. Figure 6.3.2 (b) shows the kick pattern of the *Centered* model. The red arrows in the figure show the directions of the kicking. The yellow or blue regions are the kick regions. When the defender or midfield is kicking, the rod to which the ball is rolling takes the action *LockFast*.

6.3.3 Transient States

As discussed in Chapter 5.1.1, the states of four modules of MDPs do not include all possible situations, extra transient states are still needed.

In the *Centered* model, because the opponent will be centered and still, the ball should be passed with medium speed from the defender to the midfield and from the midfield to the attacker. The six states in which the ball has medium speed are shown in Figure 6.3.3 (a). In these states, the rod, to which the ball is rolling, takes the action *LockFast*.

When the ball is slow and under control, the corresponding rod takes the action *LockSlow*. Figure 6.3.3 (b) shows these low speed and under control states.



(a) (b)

Figure 6.4: (a) Medium speed states and (b) low speed states

When the ball is moving fast, all rods of StarKick take the action *SoftBlock*. In order to change the configuration easily, these data can be loaded and saved with the MDP transition records.

6.3.4 Implementations of Opponent Models

Since there is no planning and decision process, the *Clear* model is actually an extension of the action games. It gives chances to polish the basic hand coded actions, and test the associating among these actions. For example, when the defender is kicking the ball, the midfield and "attacker" should perform the *Clear* action for a while to let the ball pass. In addition, in order to classify the current situation to the states of the MDPs, the program, which is valuable for the whole action selection, is developed and tested in *Clear* model.

The very first MDP policy iteration and reinforcement learning algorithms are constructed and tested in the *Centered* model. The reinforcement learning learns the probabilities of the dribble transitions. The policy iteration evaluates different actions and states by these transition probabilities. For the transitions which can not be learned in this model, the transition records are given manually. Policy iteration is also run on these manually configured records, and is tested by comparing the resulted optimal policy with the intended kick pattern. The configurations are seen from the results of the policy iteration algorithm, which are written into a text file with the transition records.

Although the policy iteration gets a time span to run in every cycle, because StarKick is a real time system, it may not get enough computational resources to reach a converged point. Fortunately it is a chance to hang any other thing and only do policy iteration when the ball is at the *DribbleStartStates*. Figure 6.5 shows the activity diagram of how the reinforcement learning algorithm works.

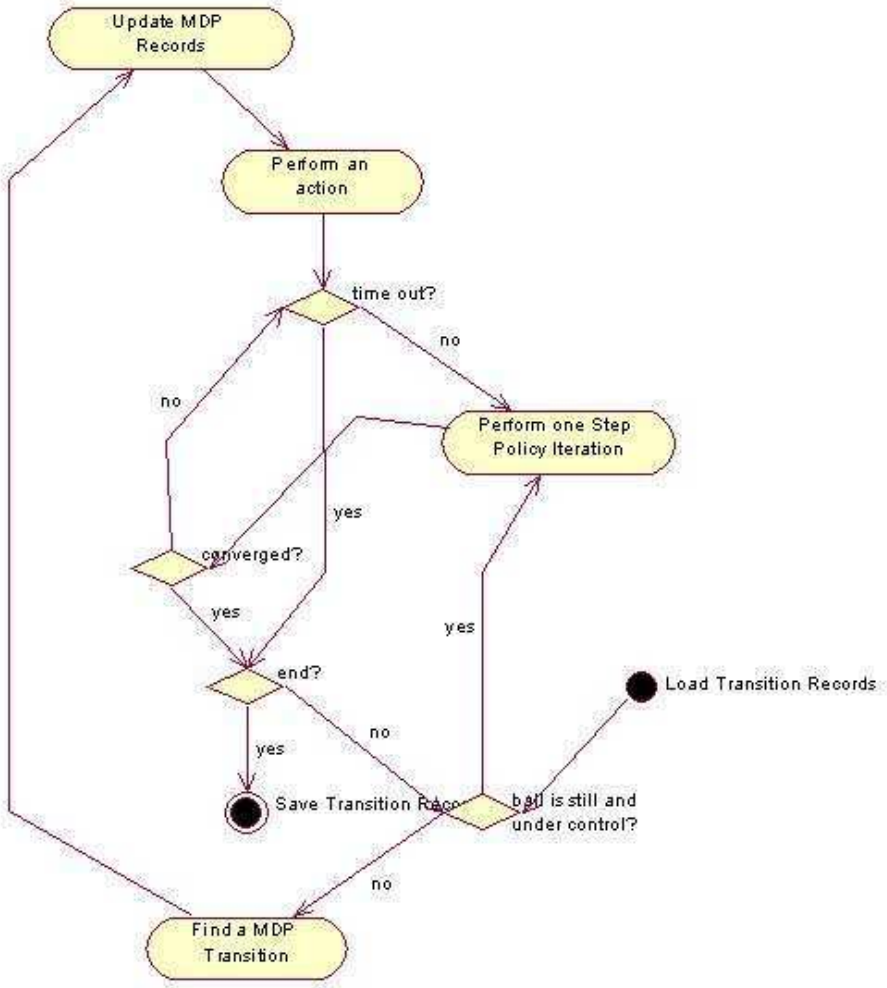


Figure 6.5: The activity diagram of the learning

At the beginning of the learning, all recorded transition numbers are set to zero. The learning program updates these records during the games. At the beginning of the policy iteration algorithm, the recorded transitions should be normalized. The activity diagram of learning Dribble module is shown in Figure 6.6.

The *KickImmediately* and *DribbleKick* module are configured in the following way. In order to avoid any *KickImmediately*, in the *Centered* model, all the kick transitions are given bad results such as the *OwnGoal*. Thus the *KickImmediately* module always chooses the *DoNothing* action as the policy. The *DribbleKick* is configured according to the kick pattern in Figure 6.3.3 (b). The intended transitions are given very good results such as the

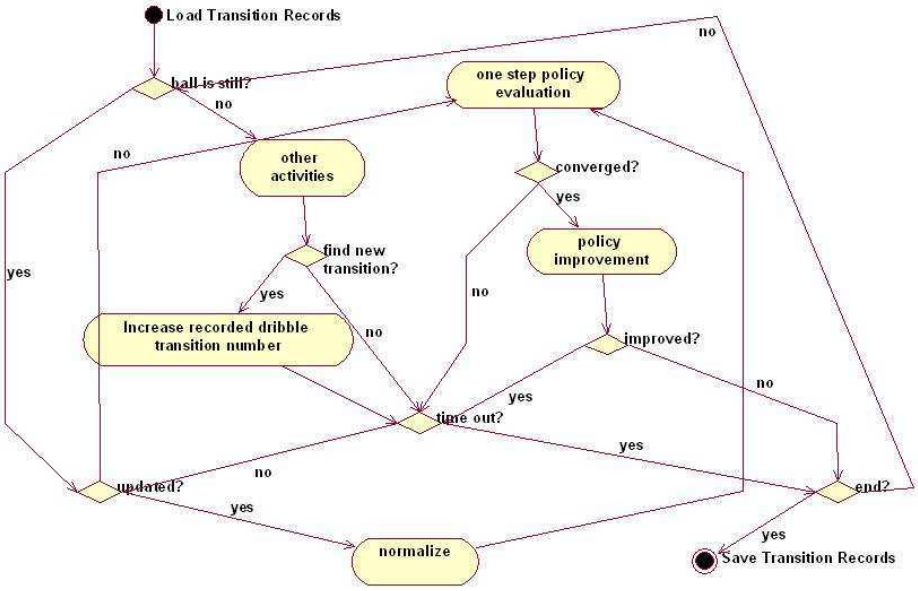


Figure 6.6: The activity diagram of learning the *Dribble* module

OpponentGoal, whereas the others are given very bad results such as the *OwnGoal*.

In the *Combined-Dribble* module, there could be some intermediate still states, which belong to the *DribbleStartStates*, and lie between two basic hand-coded actions during one *Dribble* process. Since dribble actions are regarded as the UNIT actions, these still states can not be used for any further planning. However, the learning process can be accelerated by taking these still states as the starts of other dribble transitions. Actually, during the performance of a dribble action, three or even four transitions can be updated.

In the *Single-Dribble* module, the *(Action, Target)* are taken as the dribble actions. The action is performed in the same way before the specified target is reached. For example, performing *(PassShort, DefenderLeft1Moving)* and *(PassShort, DefenderRight1Moving)* will have exactly the same effects, if the ball does not reach *DefenderLeft1Moving*, and neither *DefenderRight1Moving*, after the *PassShort* action. The learning process of the *Single-Dribble* module can be accelerated by considering all the targets after a "pass action" is performed.

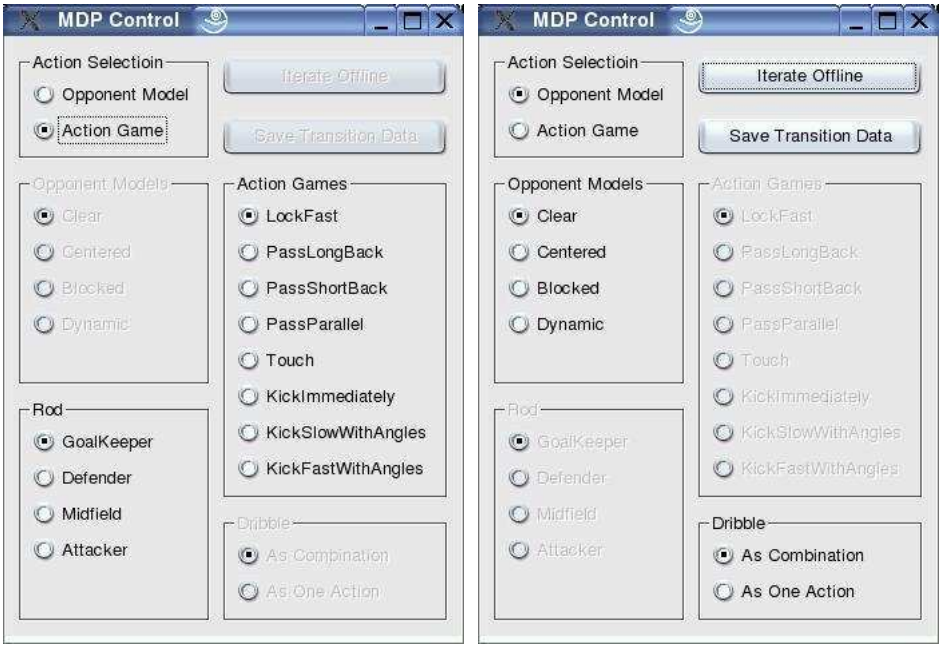
The main part of policy iteration and reinforcement learning algorithms is constructed and tested in the *Blocked* model. In addition to learning the *Dribble* transitions, there are the *Opponent* transitions, *KickImmediately* transitions, *DribbleKick* transitions, and their corresponding policy iteration processes.

The *Dynamic* model assumes that all transitions are sufficiently learned in the *Blocked*

model. The mechanism, which gives new observations more weight, is implemented in the *Dynamic* model. A log file is created to record the observations and changes of the recent policies.

6.3.5 Graphic User Interface

A GUI is constructed to switch among action games, opponent models, and dribble actions. When the "Action Game" is checked, user can choose a rod to play one of the eight predefined games, as shown in Figure 6.7 (a). When the "Opponent Model" is checked, user can choose one of the four opponent models to play against, as shown in Figure 6.7 (b). Policy iteration algorithm is run until it is converged when the button "Iterate Offline" is pressed. The learned records in the *Centered* model, the *Blocked* model, and the *Dynamic* model can be written into a data file by pressing the "Save Transition Data" button. The *Dribble* action can be configured to be *(Start, Target)* by checking the "As Combination", or to be *(Action, Target)* by checking the "As One Action", which can be found in the dribble group-box at the right-bottom corner.



(a) (b)

Figure 6.7: GUI for (a) Action games and (b) opponent bodels

Chapter 7

Experiments

We made a series of experiments for the evaluation of the performance of the basic actions and the MDP based action selection. Section 7.1 evaluates the basic actions, in which atomic actions such as *PassLongBack*, *PassShortBack*, *PassParallel*, and *LockFast*, as well as the combined action, *LockSlow* and *Touch*, are evaluated. Section 7.2 compares the performance of the different action selection models in the real games, and explains the results of the learning.

7.1 Evaluation of the Basic Actions

The *LockFast* action is evaluated in the *LockFast* game as explained in Section 6.2.2. The ball is passed from the opponent midfield to the defender of StarKick with different speed. The trials of passing are classified into different classes according to the speed of the ball. The success of a *LockFast* action is indicated by the observation that the ball is locked by the respective playing figure. Other outcomes are regarded as a failure. The successive speed classes are merged together if they have similar success rate, which results in four speed classes finally. The *LockFast* action is performed more than ten times in every situation. Figure 7.1 shows the success rate based on the results of performing *LockFast* 63 times.

The success rate of a *LockFast* action heavily depends on the speed of the ball. Figure 7.1 shows that when the speed of the ball is smaller than $500mm$ per second, every *LockFast* in the experiments can effectively lock the ball. Whereas, when the ball is very fast, which is bigger than 1300 in the experiment, the *LockFast* action rarely works. The results in this experiment are intuitive. $1300mm$ per second means that the ball travels from the opponent midfield to the defender of StarKick within 0.4 seconds. In this 0.4 seconds, StarKick needs to observe the speed of the ball at the beginning, which already takes several

process cycles to get a steady value. The time left is not enough to perform the *LockFast* action.

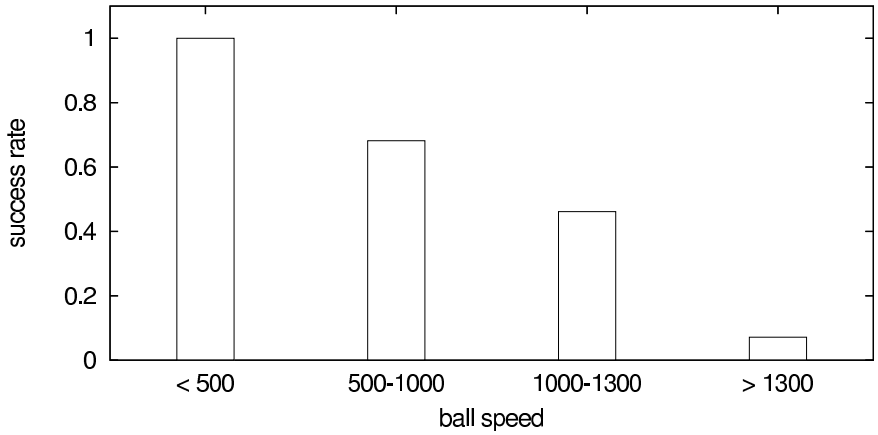


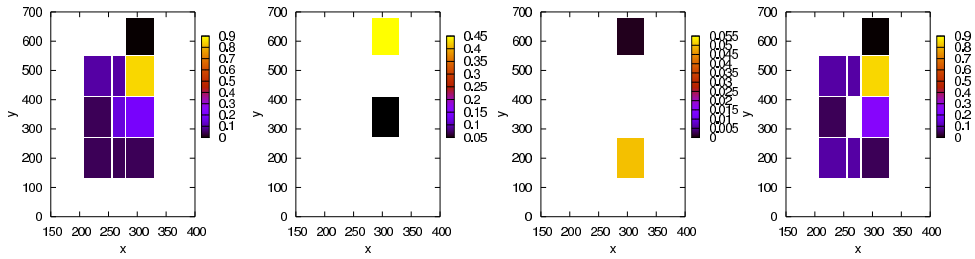
Figure 7.1: The success rate of the *LockFast* in different ball speed.

The results of the following experiments are shown in a coordinate system corresponding to the playing surface, where there are $1200mm$ along x direction and $680mm$ along y direction. The coordinate system extends the x to $1300mm$ to leave some space for the goals. The start of the x is the start of the goal of StarKick, and the end of the x is the end of the opponent goal. This coordinate system is intuitive because every region in the figures can be mapped to a region in the real plying surface.

The MDP model has the states which are defined by the different regions in the playing surface. When the ball stays in the region for less than four cycles, the action selection of StarKick can make the decision but cannot perform the decision. When the ball stays in a region from 5 to 9 cycles, the action selection of StarKick can make the decision and send the command to perform. However, the ball is going to move out of the region before the action control finishes the selected action. When the ball stays in a region for more than 10 cycles, the selected action can be successfully finished. This context is used in the evaluations of the *PassLongBack*, *PassShort*, and *PassParallel*.

A game is created to evaluate the *PassLongBack*, *PassShort*, and *PassParallel* actions. At the beginning of the game, StarKick performs *LockFast* to get the ball. Then, one of these three actions is performed. The playing figure is turned parallel to the playing surface after the passing, so that the ball moves without any further disturbing. The playing surface is divided into different regions, which are the visualization of the *DribbleStartStates* explained in Figure 5.4. The features of the passing is studied by finding out the frequency

that the ball reaches a region, and the time that the ball stays in it. The time and frequency information can be gotten by counting the region where the ball stays in every process cycle. Figure 7.2, 7.3, and 7.4 show the experiment results on *PassParallel*, *PassShort*, and *PassParallel*. In the experiments, each pass action is performed 20 times. The region where the ball is passed from is shown as the black region in the top part of Figure 7.2. Figure 7.3 and 7.4 have the same region from which the ball is passed. If the ball appears in a particular region in a process cycle, this region is recorded in the "reached regions".



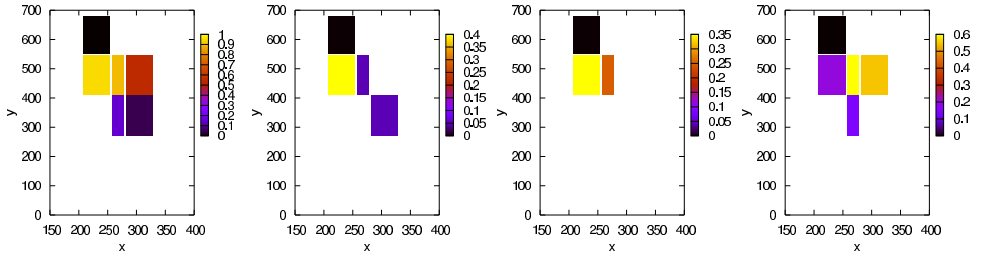
(a) (b) (c) (d)

Figure 7.2: *PassParallel* (a) reached regions, stays (b) less than 4 cycles, (c) from 5 to 9 cycles, and (d) more than 10 cycles

Figure 7.2 (a) shows the probabilities of reaching every region. Figure 7.2 (b) shows the regions in which the ball stays for less than 4 cycles after a *PassParallel*, in (c) the ball stays from 5 to 9 cycles, and the ball stays more than 10 cycles in (d). From these results, we can find that *PassParallel* passes the ball to the next region down to the one from which the passing is started. The speed of the ball after the passing is low, so that with the probability of around 90%, the ball can be locked again in the next region, which is shown as a yellow region in Figure 7.2 (a) and (d).

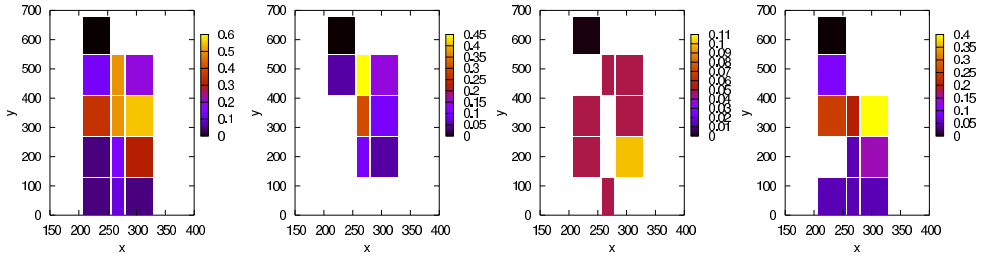
Figure 7.3 shows the evaluation of the *PassShortBack* action. From (a) and (d), we can find that *PassShortBack* can pass the ball "back" to another lock band with a success rate of around 60%. The ball passes to the next region which is below the original one, with high probabilities. However, it only stays in this region for a short time. This may cause StarKick to try some actions at this region, and these tryings will not be successful.

Figure 7.4 shows the evaluation of the *PassLongBack* action. Being different from the *PassParallel* and *PassShortBack*, *PassLongBack* passes the ball with higher speed, so that we got more regions in (b) and (c), and the reached regions almost cover all possible regions within the control range of the rod. With a success rate of around 40%, a *PassLongBack* can pass the ball from a place near the wall to a place at the center, which is shown in



(a) (b) (c) (d)

Figure 7.3: *PassShortBack* (a) reached regions, stays (b) less than 4 cycles, (c) from 5 to 9 cycles, and (d) more than 10 cycles



(a) (b) (c) (d)

Figure 7.4: *PassLongBack* (a) reached regions, stays (b) less than 4 cycles, (c) from 5 to 9 cycles, and (d) more than 10 cycles

Figure 7.4(d)

A game is constructed to evaluate the performance of *Touch* and *LockSlow*. At the start of the game, the ball is set still and in the control range of StarKick's defender. When the ball is till, but not being locked, StarKick performs *Touch*. When the ball is moving, StarKick performs *LockSlow*. When StarKick looses the control or locks the ball, the game is finished. This game is repeated 40 times and the performance of the *Touch* and *LockSlow* is evaluated by counting the successes and failures. The results are listed in Table 7.1.

A failure of a *Touch* is indicated by the ball moving out of the control range after a performance of the *Touch* action. A failure of a *LockSlow* action is indicated by the disturbance of the ball, which is happened when the figure tries to turn in a certain direction, or when the "pressing" of the lock did not happen at the right point of the time. The results shows that if the ball is still and under control, StarKick can touch the ball to lock it with a

	Success	Failure
Touch	76	3
LockSlow	29	11

Table 7.1: The evalution of the *LockSlow* and *Touch* actions

success rate of around 72.5%.

7.2 Evaluation of the MDP and RL

The very first experiment on the action selection is the implementation of the Clear model. The probabilities of the transitions are configured manually so that the game is played in the intended way. By observing the right policy being generated by the action selection, the policy iteration algorithm of MDPs is proved to be all right. Table 7.2 gives the expenses for running policy iteration algorithm separately using *(Start, Target)* and *(Action, Target)* as dribble action. The experiment is run on a PC with a $2.6GHz$ CPU and $512M$ main memory.

Dribble Action	Transitions	Policy Evaluation	Policy Improvement
(Action, Target)	201600	5.829ms	7.277ms
(Start, Target)	2700	0.356ms	0.726ms

Table 7.2: The time expenses for runing the policy iteration algorithm using the different dribble actions.

In the table, "Transitions" means the number of state-action-state triples, which is a measurement of the consumed memory space and the computing expenses of solving the MDP. "Policy Evaluation" means running one iteration on the four MDP modules for policy evaluation. "Policy Improvement" means finishing the policy improvement step of the policy iteration algorithm for the four MDP modules. It is obvious that *(Action, Target)* needs more computing resources than *(Start, Target)* because of the transitions they have. Although the *(Action, Target)* approach is much slower than the *(Start, Target)* approach, both of them perform without a problem on computing in the real table soccer games.

I have put a lot of effort into training StarKick against human beings. The training is to directly play with StarKick for more than 40 hours, as the learning algorithm updates the transitions table during the games. The transition table can be saved when the program is turned off, and be loaded again when StarKick is started next time. The thresholds for

a well-learned state action pair are set to 10 for the *KickImmediately* module, 3 for the *Dribble* module, 5 for the *DribbleKick* module, and 20 for the *Opponent* module. The *(Action, Target)* is used as the actions in the *Dribble* module. After playing with StarKick constantly for around two weeks, I found that it is too hard to train the MDP model directly in the games. For the moment, 18180 *Dribble* transitions, 3638 *KickImmediately* transitions, 491 *DribbleKick* transitions, and 1837 *Opponent* transitions are recorded. After this training, StarKick is still too awkward to play against human beings. Since the deadline of this thesis is already coming, regretfully I didn't finish the evaluation of the *Blocked* and the *Dynamic* opponent models on time. This work are described in the last chapter, hopefully they can be done in the near future. For the moment, at least the current learning records can be shown and discussed in this thesis. Figure 7.5 shows the utility values of *KickImmediatelyStartStates* and *OpponentStates*. These utilities are obtained by applying Equation 4.2 in the developed MDP and RL model. They are a measurement of how good the states are. This figure shows the different regions on the playing surface. The goal of StarKick is on the left side, the goal of the human opponent is on the right side. The three rectangles with $680mm$ as height are the *OpponentStates*, other smaller rectangles are the states in *KickImmediatelyStartStates*.

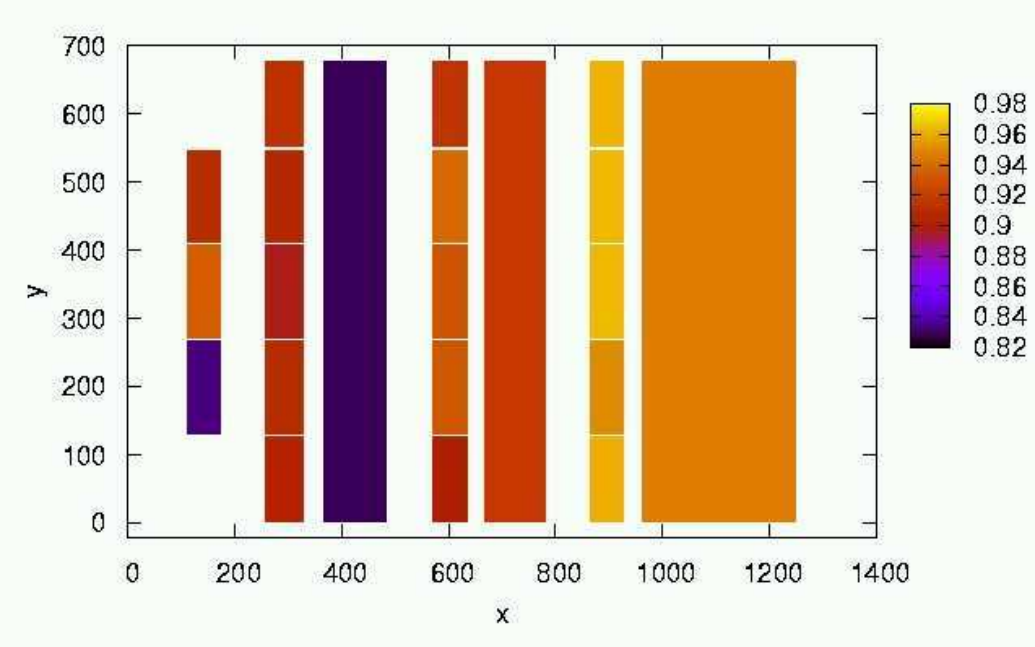


Figure 7.5: The utility of *KickImmediatelyStates* and *OpponentStates* after the learning.

In the Figure, the opponent attacker has the lowest utility value. The attacker of StarKick has the highest utility value. Interestingly, one region of the goalkeeper has a similar

utility value as the opponent attacker. Figure 7.6 (a), (b), and (c) shows the transitions on these states, so that we have a better understanding on the reasons of getting these utility values. The colors in Figure 7.6 indicates the probabilities.

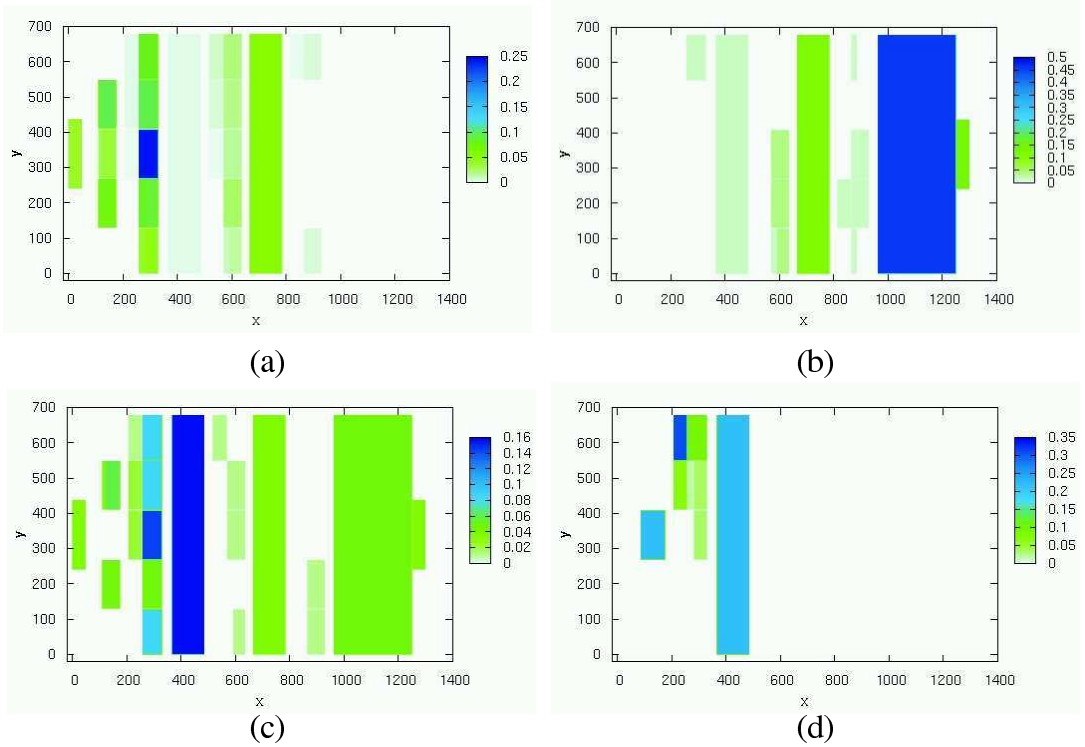


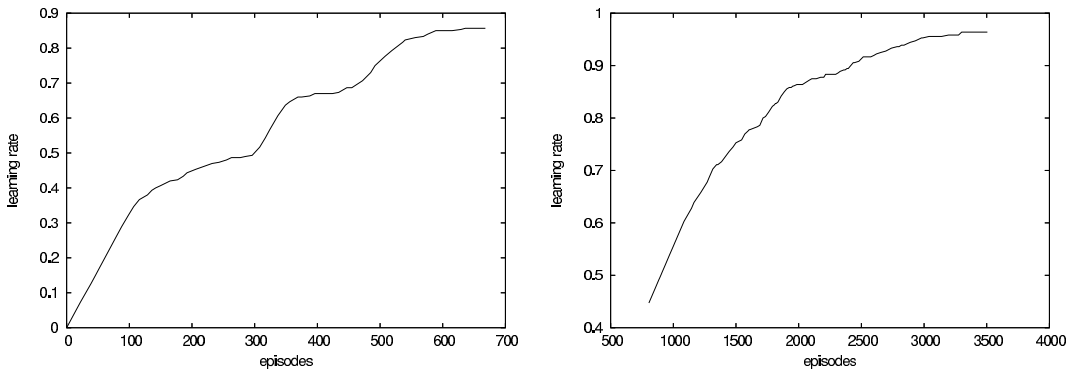
Figure 7.6: Transitions of (a) *OpponentAttacker*, (b) *KickImmediately* at *MiddleAttacker*, (c) *KickImmediately* at *LeftGoalKeeper*, and (d) *TouchLeft* at *DefenderLeft1NegativeStill*

In the Figure 7.6 (a), we can find that the opponent attacker can score a goal with the probability of around 5%. Very often, StarKick can kick the ball back immediately towards the middle defender when the opponent attacker tries to attack. The blue rectangle shows the probability of around 25%. The respective performance of StarKick at *MiddleAttacker* is worse than the attacker of the human opponent. In Figure 7.6 (b), we can see that the opponent attacker can get the ball with the probability of around 50%, if *KickImmeidately* is performed at the *MiddleAttacker*. Figure 7.6 (c) can be served as an explanation on the low utility value at *LeftGoalKeeper*. It shows that with probability of around 4%, Starkick kicks the ball to its own goal after performing *KickImmediately* at state *LeftGoalKeeper*.

The Figure 7.6 (d) shows the transitions of a touch action. At state *DefenderLeft1NegativeStill*, if *TouchLeft* is performed, the ball will stay in the same state with the probability of around 35%. The utility values of different states in the *Dribble* module are available but not

shown in this thesis, because the training for the *Dribble* module is not finished yet. There are still a lot of state-action pairs in the *Dribble* module which have never been trained in the learning. The utility values of these state-action pairs are given by the exploration function of RL, which are not a measurement of how good these situations are.

In the experiment above, the learning of the new transitions becomes more and more difficult after StarKick is trained for a week. A smaller game is constructed to show this tendency. In the game, we reduced the state space by only considering the attacker of StarKick. The *KickImmeidately* module is configured to make the fixed decision *DoNothing*. There are no *PassLongBack* actions in the *Dribble* Module. The only action in the *DribbleKick* module is the *KickFastForward*. The state in *Opponent* module is reduced to only one. The small game shares the same MDP and RL algorithms, so that it can be regarded as the sub-game of the game in the above experiment. The *(Start,Target)* and *(Action, Target)* are respectively taken as dribble actions in this game. In the *Dribble* module, the threshold for a well-learned state-action pair is set to 3. Figure 7.7 shows how fast the state-action pairs can be learned. In the Figure, the x axis is the number of performed dribble actions, the y axis is the percentages of the well-learned state action pairs of the *Dribble* module. The well-learned threshold of the *DribbleKick* module is set to 5 at the beginning. Taking *(Start,Target)* as the dribble actions, this threshold is adjusted to 8 when the recorded action number reached 300, and is adjusted to 20 when the number reached 500.



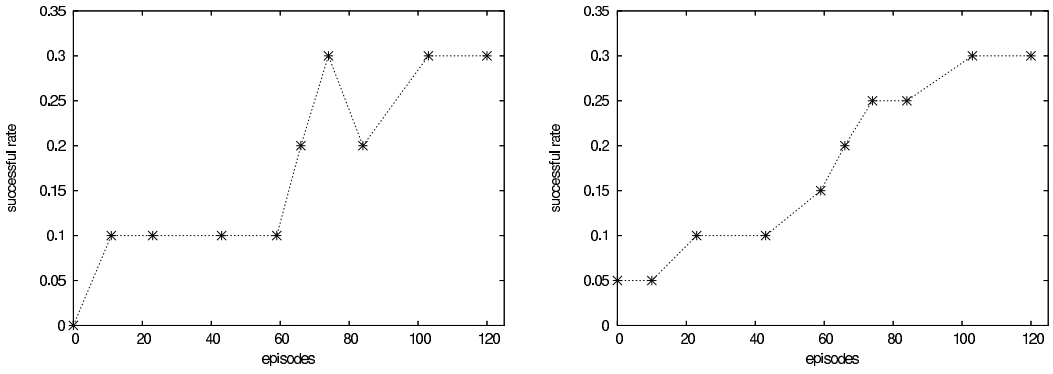
(a) (b)

Figure 7.7: The learning of (a) *Combined-Dribble* and (b) *Single-Dribble*

In this experiment, we can find that the learning of the new transitions depends on the parameters used for the MDP modules. For example, the *Dribble* module is learned faster, after the well-learned threshold of the *DribbleKick* module is increased. Figure 7.7 (a) show this phenomenon. Although Figure 7.7 (b) does not support the same conclusion, it is obvious that the chances for trying a dribble action would be less if StarKick always decides to *KickImmediately* before the ball can be stopped. Actually the *KickImmediately*

module is configured to select *DoNothing* in this experiment.

Another experiment on the same game is made to evaluate the performance of the RL over episodes. In the game, the 139 *DribbleKick* transitions learned before are used at the very beginning in the *DribbleKick* module. The *(Action, Target)* is used as the dribble action. The learning of the *Single-Dribble* module is started from zero episodes. The ball is passed from the opponent *Defender* to the *Attacker* of StartKick in the same way. A successful episode is defined as StarKick kicking the ball into the *OpponentGoal*. A failed passing is defined as StarKick loosing the ball during a passing. Ten evaluations are made in the different stages of the learning. StarKick has ten chances to pass the ball in each evaluation. Figure 7.8 (a) shows the results of these evaluations, and the results are averaged within ± 10 episodes in (b).



(a) (b)

Figure 7.8: Learning the *Single-Dribble* module in the small game (a) raw experiment data (b) averaged over ± 10 episodes

Figure 7.8 (a) shows that the success rate keeps low at the very beginning. It raise to around 0.3 after 60 episodes. The performance of StarKick from 60 to 120 episodes is much better. According to the evaluation on the atomic actions in Section 7.1, the success rate in this game should be much better. The reason for the bad performance lies in the definition of the MDP states. When the ball is near the moving limitations of a playing figure, the passing will fail if the playing figure can not be moved to the intended place. However, the results of this experiment at least shows that the RL improves the performance in this game.

The last game compares the approach in this thesis with the previous action selection using a decision tree. The ball is passed from the opponent attacker to StarKick's attacker. The games are played until StarKick scores 10 times. The numbers of kicks and the time

spent to achieve these 10 scores are recorded, which are listed in Table 7.3.

Action Selection	Time	Kick
Decision Tree	172s	67
Clear Model	286s	11
Combined Dribble	272s	10
Sinlge Dribble	339s	11

Table 7.3: The performances of different action selection models of scoring 10 times

In the table, the decision tree approach uses less time but kicks more often, because it directly maps a kick-able situation to the kick action. The *Clear* opponent model has similar performance as the *Combined-Dribble* module, because they use the same way to pass the ball to the middle when the *Combined-Dribble* module has higher rewards at the middle. The *Single-Dribble* module takes more time to achieve 10 goals because it may consist of several pass-stop cycles.

Chapter 8

Conclusion

8.1 Summary

In this thesis, basic actions such as stopping and dribbling were developed. Stopping is achieved by locking the ball between the playing surface and a playing figure. Dribbling makes the ball rolling at a controllable speed within the reachable area of the playing figures of one rod. By these new actions, the ball can be deliberately passed and stopped. A series of experiments were carried out in real table soccer games. These experiments showed that the newly developed actions are robust.

A MDP model with a reduced state space was created for the action selection of StarKick. A naive approach of using MDPs in the action selection may have a state space which consists of the grid regions, the moving speed, and the moving direction of the ball. The MDP model in the naive approach is not possible to be solved under the current computation conditions unless its state space is significantly reduced because the MDP based action selection should be executed in high dynamic environments. The approach in this thesis combined atomic actions to complex *DirbbleActions*, created different action sets according to their preconditions, and constructed four MDP modules using domain knowledge. Each module contains a set of states, and the actions that are applicable in these states. The state space of the MDP model was significantly reduced by this formalization, so that the MDP based action selection can be executed in a real-time manner.

A simple reinforcement learning algorithm was implemented in the MDP framework, in which the transition probabilities of the MDP model are updated by counting. These updates are spread by a policy iteration algorithm of the MDPs during a game. The implemented reinforcement learning algorithm maintains a list for the transitions that happened recently, gives them more weight, and is supposed to adapt the behavior of StarKick with intelligent and different human opponents.

The action selection using reinforcement learning and MDPs were evaluated in several real table soccer games. The evaluation showed that the MDP model works fine, and the reinforcement learning improves the performance of StarKick in a simplified game.

When the start states of a MDP module are the end states of another module in the same MDP, learning two modules simultaneously is slower than learning them one by one. An experiment was made to show that the "well-learned threshold" used in the *DribbleKick* module affects the learning speed of unknown state-action pairs in the *Combined-Dribble* module.

8.2 Future Work

Although the reinforcement learning algorithm improved the performance of StarKick in a simplified game, the full game was not possible to be evaluated in this thesis because the learning process took too much time (more than 40 hours playing in real games).

Several ideas can be used to make a better learning process in the future work. The first one is to study the probabilities of the atomic actions, and copy the results to corresponding places in the transition table. For example, if the *LockSlow* action is performed after the pass actions in most cases, as it happens in the approach of this thesis, the evaluation of the atomic pass actions can be used to approximate the probabilities of the respective *Dribble* transitions. The learning process can be significantly reduced in this way.

The second one is to isolate the module if the actions have to be evaluated in the game, so that each module is learned faster. For example, the *KickImmediately* module is turned off so that the *Dribble* module will have more chances to dribble a ball.

The third one is that a pure exploration stage of the reinforcement learning in the real games with human opponents should be avoided in practice. A pure exploration stage in real table soccer games takes a lot of time, and human opponents will react differently when StarKick keeps on trying some stupid actions for a long time. The situation would be much better if StarKick can play in a competitive way before it learns from the transitions in the full games, while the life-long learning improves its performance for different human players. If the first and second ideas can be served as the effective ways to initialize a transition table, the *Dynamic* model implemented in this thesis should adapt the performance of StarKick to different human players. The problem left is how the reinforcement learning explores a state-action pair whose transition probabilities are unknown. A simple way to do this is the exploration is preferred to with a probability, while the exploitation is performed in the other cases.

The MDP approach in this thesis uses a *Dribble* module for different *PassActions*. This causes extra work in the learning because the *Touch* action has different preconditions from other pass actions. Although the four modules used in this thesis significantly reduce the impossible state-action pairs, there is still a space to improve them further. In the future work, the *Touch* action should be identified from other pass actions by modules, so that the learning and playing can be more effective.

The MDP states in this thesis are created in a simple way by dividing the playing surface into five different regions along the y direction. There are problems when a figure reaches the end of its moving range. In the future work, the MDP states should be refined by integrating more domain knowledge. For example, there should be a state at the end of the moving range, where a figure can only pass the ball in one direction.

Although the developed basic actons are robust and already very different from the previous approaches, several improvements are possible in StarKick in the future.

The first one is to switch to the second playing figure when the ball is locked by the first one, and is within the reachable area of the second playing figure. A MDP module using this action should have a state which consists of the information of the playing figures.

The *KickActions* in this thesis appear too slow to score in a game against an advanced human player. In the second part of the suggested work, a set of kick actions which can challenge the advanced human players should be developed.

The third point is that two specific action sets for the goalkeeper and the midfield should be developed because the goalkeeper is very close to the own goal and the midfield has a very limited moving range.

The final point is that any turning should turn in a “positive” direction in order to avoid that the ball is kicked towards the own goal by accident.

Appendix A

Phrases

Playing surface: The miniature of a real soccer field, as the ground of a table soccer game.

Rod: The sticks installed over the playing surface, by which players can control the figures and playing. Rods are named as goalkeeper, defender, midfield, and attacker by there position. There are four rods controlled by human beings, as well as the other four by StarKick.

Playing figure: The single human miniature installed along the rod. There are one figure on goalkeeper, two on defender, five on midfield, and three on attacker. Only the bottom of the playing figures can touch the ball.

Lock Ball: Stop the ball by pressing. If the ball is locked, it is stuck between the playing surface and the playing figure in the lock band.

Lock Band: The rectangle areas on the playing surface, in which the ball can not be kick in one direction. Only *Lock* action is possible in this band in one direction.

Negative Lock Band: The lock band is at the negative direction with respect to a certain rod. Negative direction is towards the player's own gate.

Positive Lock Band: The lock band is at the positive direction with respect to a certain rod. Positive direction is towards the opponent's gate.

Kick-able Band The rectangle areas in which the ball can be kicked out without any problem in both directions.

Opponent Model: A set of assumptions on the behavior of the opponent. There are four opponent models in this thesis.

Action Games: The small program designed particularly for practicing and refining some actions. They are useful on debugging the hand coded basic actions.

Kick Pattern: A map (soccer field) in which all kick actions are shown at the places where they are going to happen.

Kick Region: A rectangle area in which the playing figure take a specific kick action.

Opponent: StarKick is a game requires two playing sides, human players versus StarKick. In this thesis, human players is always called Opponent.

Own Side: The side controlled by StarKick.

Right, Middle, and Left of the Rod: With respect to the coordinate system of StarKick, right is the direction of negative y axis; middle is the areas near the zero; left is the direction of positive y axis.

Dribble: Pass the ball between figures in the same rod. The dribble action in the thesis is always started at a still state, end ended with a kick action.

Appendix B

Contents of the Appendent CD

Some videos, which record how StarKick is playing, are included in the appendent cd. Besides, all the C++ source code related to this thesis, latex source code to make the thesis, and experiment data are also included in the appendent CD.

The follows are the list of the directories and files in the CD.

cd:/video video records on *LockFast*, *LockSlow*, *KickFast*, *KickSlow*, *KickImmediately*, *PassLongBack*, *PassShortBack*, *PassParallel*, *Touch*, and some games.

cd:/thesis.pdf: The electronic version of the thesis.

cd:/SourceCode/Program/
mdpState.cpp and mdpState.h: implement the reinforcement learning and the MDP model.
actionSelectionMdp.cpp and actionSelectionMdp.h: are action selection program of StarKick
actionControlMdp.cpp and actionControlMdp.h: implement the basic actions. mdpControlWindow.cpp and mdpControlWindow.h: are the GUI.
The program need to be run in StarKick, which is not possible to be included here. The related source codes are sum to 15156 lines.

cd:/SourceCode/Thesis
All source files for making the thesis document are included in this directory.
plot: includes all raw data to make the present graphs.
figures: includes all images appeared in the thesis.
doc: are some reference readings.

cd:/ExperimentData

MdpDataBlocked.txt: the learning records for the whole game.
MdpDataClear.txt: the configuration for the Clear model.
MdpDataCentered.txt: the configuration for the centered model
record/: data which record the different learning stage.

List of Figures

2.1 The pictures of (a) KiRo. (b) StarKick. 4
2.2 Software Architecture of KiRo. 5
2.3 Camera view of (a) KiRo and (b) StarKick. 5
2.4 The Visualizations of World Model of (a) KiRo and (b) StarKick 6
2.5 Coordinate System. 6
2.6 Action Control of KiRo [11] 7
2.7 Action Selection of KiRo [11] 8
2.8 One Iteration of the Planning [9] 9

3.1 Action Sets of StarKick 11
3.2 (a) Negative Lock Band (b) Positive Lock Band 12
3.3 The situations of (a) *LockFast* and (b) *LockSlow* 13
3.4 the oscillation of the rods after a turning 14
3.5 The activity diagram of the *LockFast* 15
3.6 The three directions of the *Kick* 16
3.7 (a) *PassShortBack*, *PassLongBack*, *PassParallel*, and (b) *Touch* 17
3.8 (a) The motor current of the *PassLongBack* and (b) the situations of the *PassParallel* 17

4.1 The definition of the MDPs [2] 20
4.2 Value iteration algorithm for calculating the utilities 21
4.3 Policy iteration algorithm [1] 22
4.4 Adaptive Dynamic Programming with a finite search space MDP model . . 23
4.5 Generic Dyna algorithm [8] 24
4.6 Simplistic comparison of the architectures: (a) reactive systems, (b) conventional planning, and (c) Dyna architecture [8] 25

5.1 (a)The *PassActions* in the planning and (b) the complex *DribbleActions* in the planning 28
5.2 State sets of the MDPs. 29
5.3 Kick-able regions 30

5.4 *DribbleStartStates* 31
5.5 Take the basic actions in the *Dribble* module 32
5.6 The applicable *Combined-Dribble* actions in one state 33
5.7 Activity diagram of the dribble action *(Action, Target)* 34
5.8 An example of the *(Action, TargetList)* 35
5.9 Action sets of the MDP model 35
5.10 MDP modules in the "playing" process (a) *(Start Target)* as the *DribbleActions* and (b) *(Action, Target)* as the *DribbleActions* 36
5.11 *KickEndStates* 37
5.12 The transitions of the *(Start, Target)* dribble 38
5.13 The *DribbleEndStates* of the *Single-Dribble* module. 39
5.14 Normalizing the *Kick* transitions in the *Dynamic* model 41

6.1 (a) Component of KiRo Behavior System. (b) Layers. 44
6.2 The "control" program 47
6.3 The kick patterns of the (a) *Clear* model and (b) *Centered* model 48
6.4 (a) Medium speed states and (b) low speed states 49
6.5 The activity diagram of the learning 50
6.6 The activity diagram of learning the *Dribble* module 51
6.7 GUI for (a) Action games and (b) opponent bodels 52

7.1 The success rate of the *LockFast* in different ball speed. 54
7.2 *PassParallel* (a) reached regions, stays (b) less than 4 cycles, (c) from 5 to 9 cycles, and (d) more than 10 cycles 55
7.3 *PassShortBack* (a) reached regions, stays (b) less than 4 cycles, (c) from 5 to 9 cycles, and (d) more than 10 cycles 56
7.4 *PassLongBack* (a) reached regions, stays (b) less than 4 cycles, (c) from 5 to 9 cycles, and (d) more than 10 cycles 56
7.5 The utility of *KickImmediatelyStates* and *OpponentStates* after the learning. 58
7.6 Transitions of (a) *OpponentAttacker*, (b) *KickImmediately* at *MiddleAttacker*, (c) *KickImmediately* at *LeftGoalKeeper*, and (d) *TouchLeft* at *DefenderLeft1NegativeStill* 59
7.7 The learning of (a) *Combined-Dribble* and (b) *Single-Dribble* 60
7.8 Learning the *Single-Dribble* module in the small game (a) raw experiment data (b) averaged over ± 10 episodes 61

List of Tables

7.1 The evalution of the *LockSlow* and *Touch* actions 57
7.2 The time expenses for runing the policy iteration algorithm using the different dribble actions. 57
7.3 The performances of different action selection models of scoring 10 times . 62

Bibliography

[1] Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–107, 2000.

[2] Hector Geffner. Modelling Intelligent Behaviour: The Markov Decision Process Approach. In *IBERAMIA*, pages 1–12, 1998.

[3] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[4] S. Keerthi and B. Ravindran. A tutorial survey of reinforcement learning, 1995.

[5] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa, and H. Matsubara. Robocup: A Challenge Problem for AI. In *AI Magazine*, 1997.

[6] B. Nebel, T. Weigel, and J. Koschikowski. Tischfussball, Hockey oder dergleichen und Verfahren zur automatischen Ansteuerung der an Stangen angeordneten Spielfiguren eines Tischspielgeräts für Fussball-, Hockey- oder dergleichen. In *Patent DE 102 12 475*, Deutsches Patent-und Markenamt, Januar 2005.

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

[8] Richard S. Sutton. Dyna, an Integrated Architecture for Learning, Planning and Reacting. In *Working Notes of the AAAI Spring Symposium on Integrated Intelligent Architectures*, Waltham MA 00245, 1991.

[9] M. Tacke, T. Weigel, and B. Nebel. Decision-Theoretic Planning for Playing Table Soccer. In *Proceedings of the 27th German Conference on Artificial Intelligence*, pages 213–225, Ulm, Germany, 2004.

[10] T. Weigel. Kiro – A Table Soccer Robot Ready for the Market. In *Knstliche Intelligenz Heft 01/05*, 2005.

[11] T. Weigel and B. Nebel. Kiro – An Autonomous Table Soccer Player. In *Proceedings of RoboCup Symposium '02*, pages 119 – 127, Fukuoka, Japan, 2002.

[12] T. Weigel, D. Zhang, K. Rechert, and B. Nebel. Adaptive Vision for Playing Table Soccer. In *Proceedings of the 27th German Conference on Artificial Intelligence*, pages 424–438, Ulm, Germany, 2004.