RoboCup 机器人足球队 2D 仿真组高层策略的设计

摘要：

本论文介绍了 RoboCup 机器人足球队 2D 仿真组的设计和实现，完成了一个完整的符合 RoboCup2D 仿真组比赛要求的机器人足球队。首先介绍了 RoboCup 和其仿真组比赛的背景，接着介绍了仿真组比赛的环境。从第三章开始介绍了智能体(agent)的基本结构和球队设计的思想，并重点阐述如何设计队员的各种场上能力，各球员间的配合策略，以及守门员的防守策略。此外还完成了阵形的实现和转换，以及基本的场上教练的设计。最后通过比赛对所完成的球队和使用的策略进行了分析和评估。

关键词　RoboCup；仿真组；智能体；配合策略

High-level Strategy Design of RoboCup 2D Simulation Soccer Team

Abstract

In this thesis the design and implementation of a RoboCup 2D simulation soccer team is introduced and the aim to design an integrated soccer team which is qualified for the standard RoboCup 2D simulation competition is fulfilled. First, the background of RoboCup and its simulation league is presented followed by the introduction of the competition environment of simulation league. In chapter three, the basic structure of an agent and the framework of the team are described. Then how to design the skills of a soccer player and cooperation strategy between players in the field and the goalie strategy are particularly presented. Simultaneously, the team formations and transformation between different formations in a match are implemented. Besides, a rudimentary on-line coach is designed. In the last chapter the analysis and evaluation of the strategy used in current team are carried out through actual matches.

Keywords: RoboCup, Simulation League, Agent, Cooperation strategy

目录

1 绪论

1.1 RoboCup 简介

机器人足球世界杯(RoboCup) 是国际上一项为促进分布式人工智能、智能机器人技术及其相关领域的研究与发展而举行的大型比赛和学术活动。它通过提供一个标准的比赛平台来检验各种智能机器人技术。它的最终目标是能在 2050 年开发出能击败人类足球运动员的机器人足球队。RoboCup 比赛方式有多种，总体可以分为两大类，软件模拟(仿真组比赛)和实物机器人比赛。

机器人足球世界杯仿真组比赛(RoboCup Simulation League)是使用纯软件仿真的虚拟足球队之间的比赛，不考虑实际硬件的复杂性。目前根据标准比赛平台的不同分为 2D 组和 3D 组。2D 组比赛即是在二维环境下进行的比赛, 而 3D 组则在此基础上考虑了立体环境因素。2D 组比赛上下半场各 3000 个时间单位，并且中场时不交换场地。

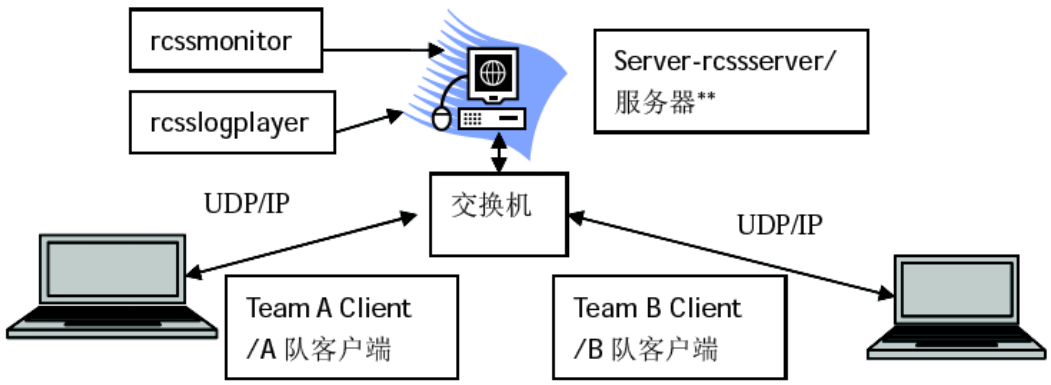
1.2 仿真组比赛简介

RoboCup 仿真组的比赛提供了一个分布式控制的多智能体(Multi-Agent)环境，在这样的环境下可以进行智能体(agent)结构的设计和测试，策略算法的研究和实现，以及各种人工智能理论的学习和应用。比赛在一个标准局域网或独立的计算机系统下进行，采用 Client/Server 结构。RoboCup 联合会提供了一个标准 Server 系统 rcssserver，参赛球队各自编写 client 程序参加比赛。rcssserver 模拟了实际足球比赛场景中大部分的特性,包括双方队员、球、标准

的场地、以及与人类足球比赛相似的规则。除此之外还引入了真实世界中的复杂性，比如队员感知信息的不确定性、队员体力的有限性、各队员之间交流的有限性、队员执行动作的不确定性和执行动作成功的概率性。

RoboCup2D 仿真组比赛结构如图 1.1：

图 1.1 2D 仿真组比赛结构



注：rcssmonitor 和 rcsslogplayer 不一定要运行在 server 上，实际上可以运行在该局域网内任何一台计算机上。

在开发调试过程中为了方便起见，rcssserver 和两个 client 可运行在同一台计算机上。

2D 仿真组的比赛场景如下:

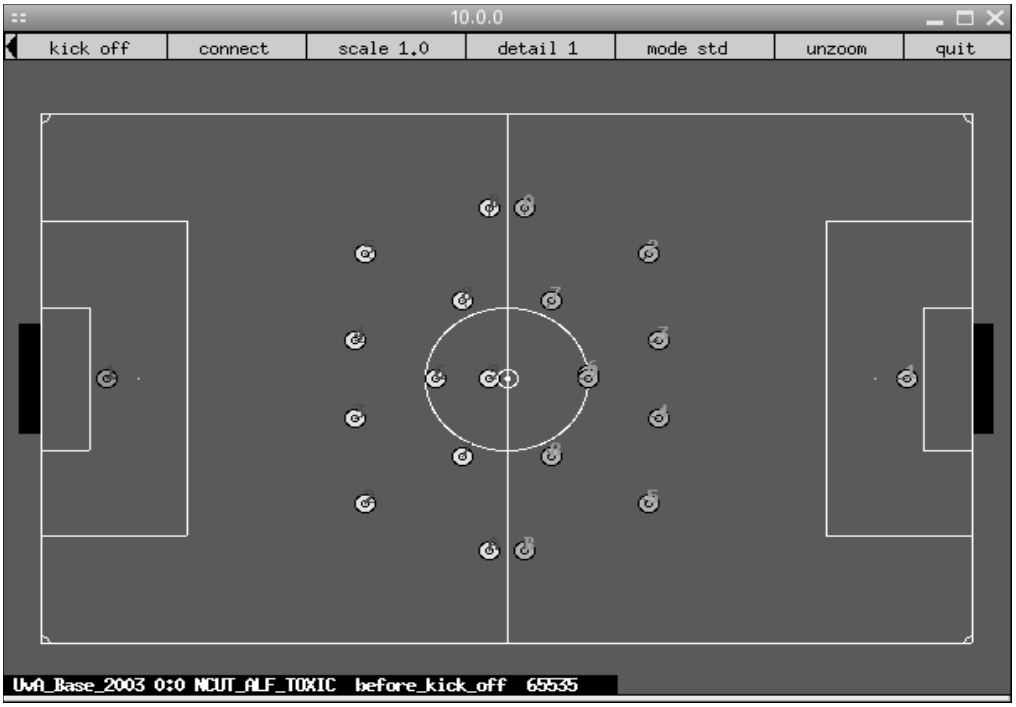


图 1.2 2D 仿真组的比赛场景

2 球队程序的开发环境和基本结构

2.1 开发环境

由于 RoboCup 仿真机器人足球队程序运行在 Linux 操作系统下、采用 Client/Server 结构、多进程等的特点，因此在球队程序的设计中采用 Ubuntu Linux 5.04 作为开发平台，使用标准 C++语言进行程序的实现，同时使用 GNU 的 C++编译器 g++(gcc)3.3.5 作为编译器，并采用 Makefile 管理整个工程同时使用 IDE 工具 Anjuta 方便工程的管理。

2.2 球队的基本结构

球队由 11 个 agent 进程和 1 个 coach 进程组成，coach 进程是场上教练，其设计见 8.2。每一个 agent 进程的程序是相同的，通过使用 shell 脚本启动整个球队。因此所需要设计的是 1 个 agent 程序，而这个程序具有多进程的特性，允许使用 shell 脚本启动该程序自身 11 次来组成 11 个场上队员。

每个 agent 的结构如图 2.1：

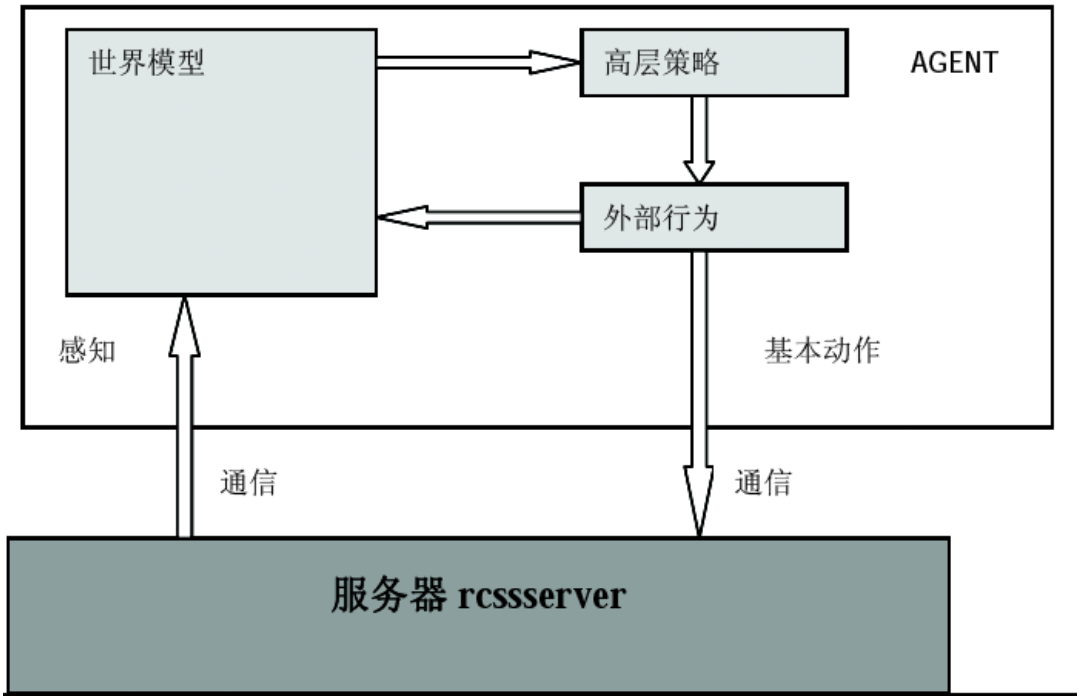


图 2.1 agent 的结构

进行仿真比赛时,每个 agent 要维护一个世界模型，通过世界模型 agent 才能感知当前球场上的情况。高层策略则是通过对世界模型的分析来决定如何与其它 agent 相配合。当高层策略作出后，agent 会进行相应的底层行为，具体表现为 agent 为了实现高层的决策与服务器进行通信，来进行各种基本的动作，比如转身、跑位、带球、截球、传球、射门等。同时，agent 将根据自己的外部行为来对自己的世界模型进行更新。

Agent 的设计参考 UvA TrilearnBase2003 球队的程序结构，并在世界模型，

与 server 通信的策略上采用其公开的源码，并通过参考其部分低级动作（跑、踢球等）源码来设计中高层动作（断球，传球等）。

在高层球员间配合策略的方面，采用基于角色分配的配合方案，并实现比赛中阵形的转变。

2.3 球队启动的 shell 脚本

球队的启动执行 start.sh 脚本文件来进行，该脚本文件的主要作用是启动 11 个 agent 进程，脚本的流程图如图 2.2：

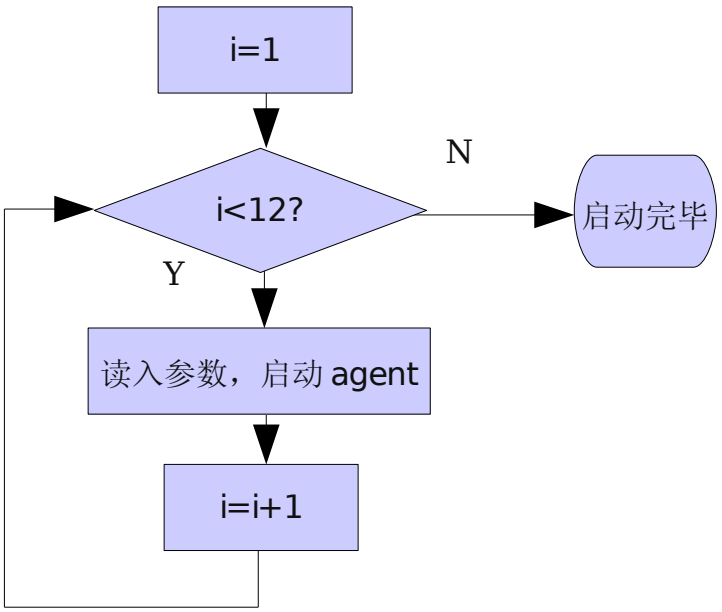


图 2.2 球队的启动

start.sh 脚本参考了 TrilearnBase2003 球队的启动脚本，使用 tcsh 脚本语言书写。

3 Agent 程序的结构

3.1Agent 各功能模块

根据图 2.1 所展示的 Agent 组成结构，设计不同的类来实现各部分的功能。世界模型由 WorldModel 类实现，球员由 BasicPlayer 类和 Player 类实现，感知由 SenseHandler 类实现，基本动作通过 ActHandler 类实现，而所有的通信则由 Connection 类实现。而高层策略则由 Player 类中的 toxic()方法和 toxic_goalie()方法实现。

以上各类的功能的实现，需要其它许多辅助功能类的配合。WorldModel 类需要通过 Object, FixedObject, DynamicObject, BallObject, PlayerObject, AgentObject 类得到球场上所有物体的各种信息，通过 VecPosition 类表示场上的坐标。Player 类通过 Formations, PlayerTypeInfo, FormationTypeInfo 类得到阵形的信息并控制阵形，通过 Stamina 类实现队员的体力。BasicPlayer 类必需的参数由 PlayerSetting 类提供。SenseHandler 类通过 Parse 类将 rcssserver 发送的信息读入世界模型。ServerSettings 类储存了 Parse 类转换过的 rcssserver 信息。Parse 类负责将 rcssserver 可接受的基本命令以及发送的信息给格式化，使更容易被理解并被其它类引用。Circle, Geometry, Line, Rect 类提供了基本的几何计算功能，Logger 类提供了对比赛进行记录的功能，Time 类提供了整个球队的时钟，Timer 类则提供了计时器的功能。

各类的关系如图 3.1

3.2Agent 的启动和类的连接

主程序源文件 main.cpp 读入 agent 启动时需要的各种参数，并启动各个类模块。该源文件所执行的操作见流程图 3.2。

主程序源文件 main.cpp 中对 Formations, WorldModel, Connection, ActHandler, SenseHandler, Player 类进行了调用。这几个主要的类自身对其它的辅助功能类进行调用，进而实现所有类的连接。

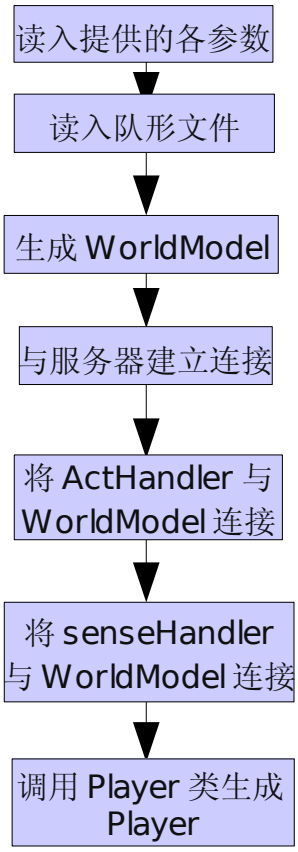


图 3.2 Agent 的启动流程图

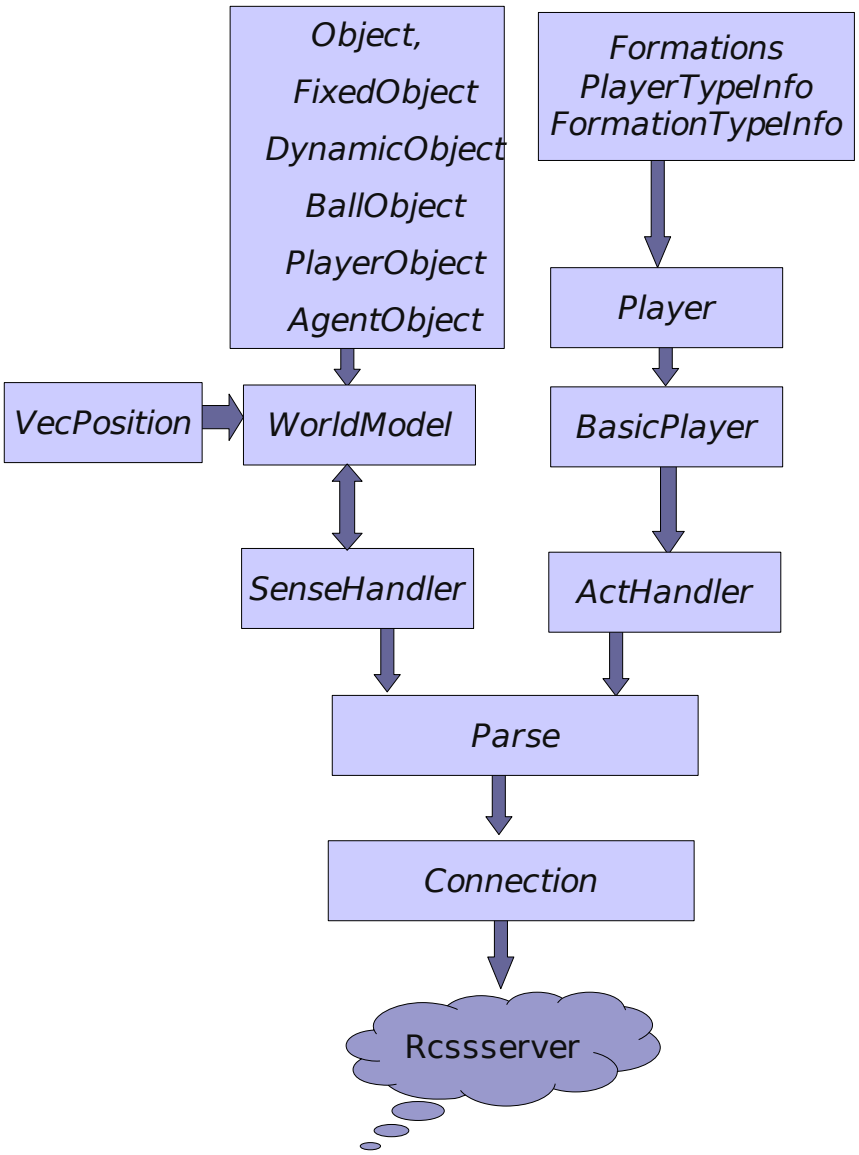

Object,
FixedObject
DynamicObject
BallObject
PlayerObject
AgentObject
Formations
PlayerTypeInfo
FormationTypeInfo
Player
VecPosition
WorldModel
BasicPlayer
SenseHandler
ActHandler
Parse
Connection
Rcssserver

图 3.1 各类的关系

4 球员底层动作的设计

4.1 底层动作的介绍

球员的底层动作是指球员的基本感知能力和个人能力，比如在赛场上寻找球，得到自己在赛场上的位置，调整自身状态等等。

4.2 底层动作的实现

4.2.1 实现原理

底层动作实现的基本原理是如下：从 WorldModel 类中读出相应的数据，然后进行处理，将输出数据赋给 SoccerCommand 类，进而完成相应的动作。不仅是底层动作，中层和高层动作也是基于这个基本原理来实现的。

所有的底层动作均为 BasicPlayer 类中的方法。

主要的底层动作方法见表 4.1：

表 4.1 BasicPlayer 类中底层动作方法列表

方法名称	实现功能	解释
alignNeckWithBody()	使球员脖子与身体匹配	
turnBodyToPoint()	将身体正面转向给出的位置	
turnBackToPoint()	将身体背面转向给出的位置	用于守门员
turnNeckToPoint()	将脖子转向给出的位置	
searchBall()	寻找球的位置	
dashToPoint()	向某个位置加速	
teleportToPos()	直接将球员移动至指定的位置	不能在比赛进行中的时候使用
turnBodyToObject()	将身体正面转向给出的物体	
turnNeckToObject()	将脖子转向给出的物体	

方法名称	实现功能	解释
freezeBall()	停球	

这里主要介绍 searchBall 方法和 dashToPoint 方法的具体实现。

4.2.2 searchBall 搜索球方法的实现

searchBall 方法使球员在看不到球的情况下试着去寻找球。该方法返回一个转身命令，球员将身体转动一个等同于其目前视觉圆锥体范围的角度。这样球员将得到在与刚才完全不同的方向上的新的视觉范围，进而增大了在下一个时间周期中看到球的可能性。

球员的可视范围(visible_range)见图 4.1

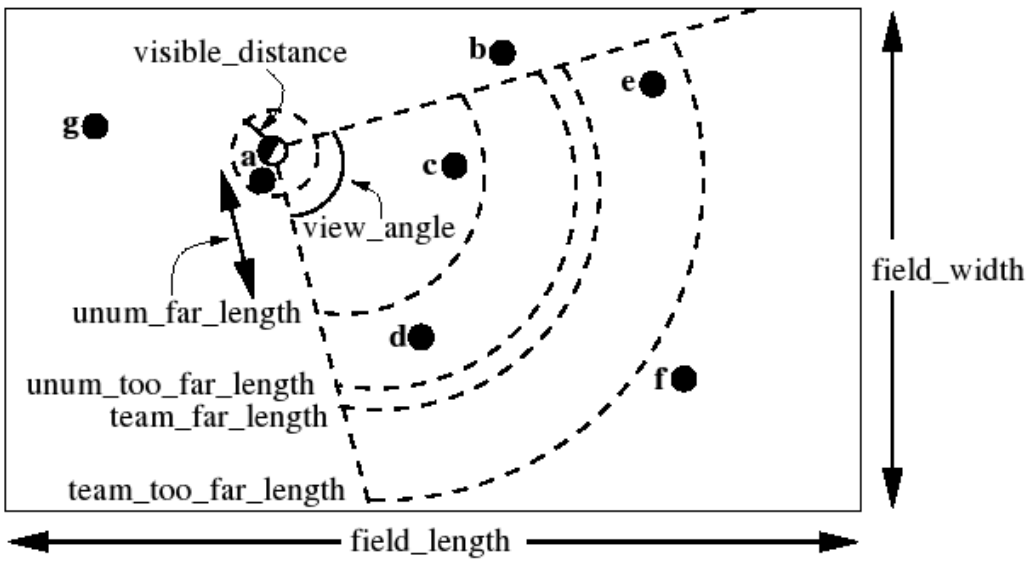


图 4.1 球员的可视范围

可视范围由可视角度(visible_angle)和可视距离(visible_distance)构成，如图 4.1，在球员正面的可视角度内，根据物体离球员距离 dist 的不同，物体被球员捕捉到，即被看到的概率不同，并且得到的信息完整度不同。

如果 dist ≤ unum_far_length，球员号码和球队名称总是可见。

如果 umum_far_length < dist < unum_too_far_length，球队名称总是可见，但是球员号码的可见概率随着 dist 的增加从 1 到 0 线性降低。

如果 dist ≥ unum_too_far_length,球员，球员号码将变得不可见。

如果 dist ≤ team_far_length，球队名称总是可见。

如果 team_far_length < dist < team_too_far_length，球队名称的可见概率随着 dist 的增加 1 从 0 线性降低。

应用 searchBall 方法的结果并不仅仅只增大了寻找到球的概率，同时也增大了看到其它队员，包括己方球员和对方球员的可能性。但是由于该方法主要运用于寻找球，因此命名为 searchBall 方法。

searchBall 方法的流程如图 4.1

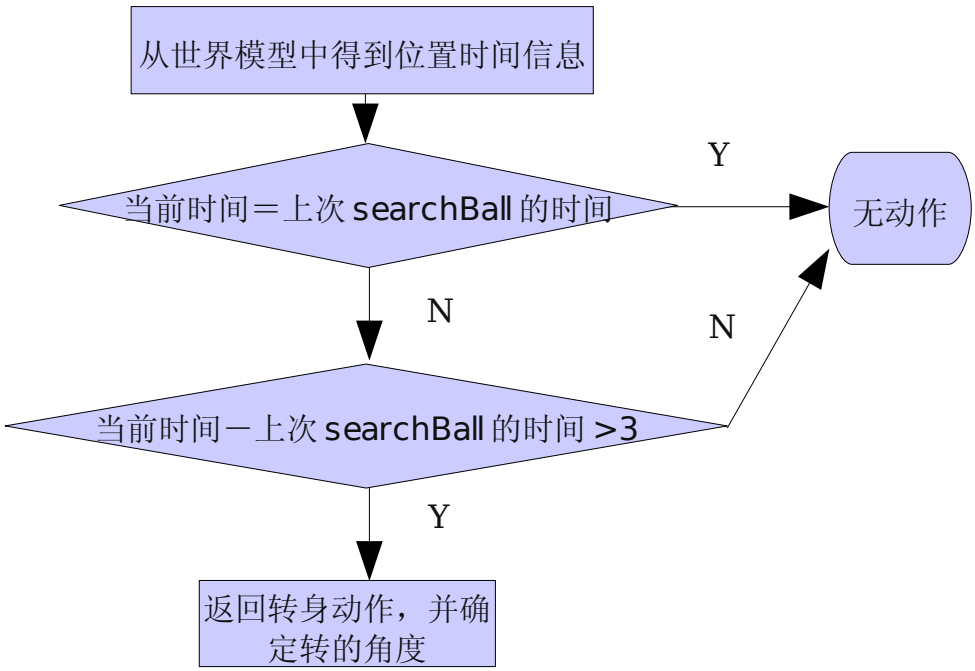


图 4.1 searchBall 方法流程图

4.2.3 dashToPoint 跑位方法的实现

dashToPoint 方法生成一个奔向给定位置的动作。该方法接受一个位置参数，然后返回一个动作命令。队员通过这个方法可以尽可能的接近某个给定的位置。

首先通过 WourlModel 的 getPowerForDash 方法得到奔跑的速度，然后通过赋值给 SoccerCommand 类产生该动作。

该方法的实现代码如下：

```
SoccerCommand BasicPlayer::dashToPoint( VecPosition pos, int iCycles )
{
  double dDashPower = WM->getPowerForDash(
                                          pos - WM->getAgentGlobalPosition(),
                                          WM->getAgentGlobalBodyAngle(),
                                          WM->getAgentGlobalVelocity(),
                                          WM->getAgentEffort(),
                                         iCycles);//从世界模型中得到奔跑的速度
  return SoccerCommand( CMD_DASH, dDashPower );//产生以 dDashPower 为
                                                //奔跑速度的 Dash 的命令
}
```

5 球员中、高层动作的设计

5.1 中高层动作的介绍

球员的中高层动作主要指球员间相互配合的基本能力，包括传球，带球，带球转身，断球等，因此是球员间相互合作的基础。高层配合策略的实现以及实现效果依赖中高层动作的实现。

5.2 中高层动作的实现

5.2.1 实现原理

中高层动作实现采用的基本原理与 4.2.1 完全相同，即从 WorldModel 类中读出相应的数据，然后进行处理，将输出数据赋给 SoccerCommand 类，进而完成相应的动作。不同的是中高层动作从 WorlModel 类中读取了种类更多，数量更大的信息，并进行了更复杂的处理，进而实现了较复杂的中高层动作。

所有的中高层动作均为 BasicPlayer 类中的方法。

主要的中高层动作方法见表 5.1：

表 5.1BasicPlayer 类中中高层动作方法列表

方法名称	实现功能	解释
kickTo()	将球踢向给定的位置	
kickBallCloseToBody()	在踢球的过程中将球控制在离身体一定的距离内	
dribble()	带球	
turnWithBallTo()	带球转身	
interceptClose()	断球	
directPass()	传球	
leadingPass()	预设提前量的传球	

这里主要介绍 dribble 方法，intercept 方法，directPass 方法的实现。

5.2.2 dribble 带球方法的实现

dribble 方法使球员具有带着球运动的能力，具体是指球员带球运动，同时将球保持在离自己身体一定距离范围内。dribble 动作的目标是能够以一定的速度不断的踢球，直到将球踢至所要求的地点。

dribble 方法一共使用两个参数 ang 和 dribbleT。ang 代表带球前进的角度，dribbleT 代表带球的方式。dribbleT 参数一共有三种：DRIBBLE FAST，DRIBBLE SLOW，DRIBBLE WITH BALL。

DRIBBLE FAST：球员以快速方式带球前进，在带球过程中球离球员的距离相对较远。

DRIBBLE SLOW：球员以慢速方式带球前进，球离球员的距离比快速带球方式较近。

DRIBBLE WITH BALL：安全带球方式，速度最慢，但是球离球员的距离非常的近。

dribble 方法执行流程如图 5.1：

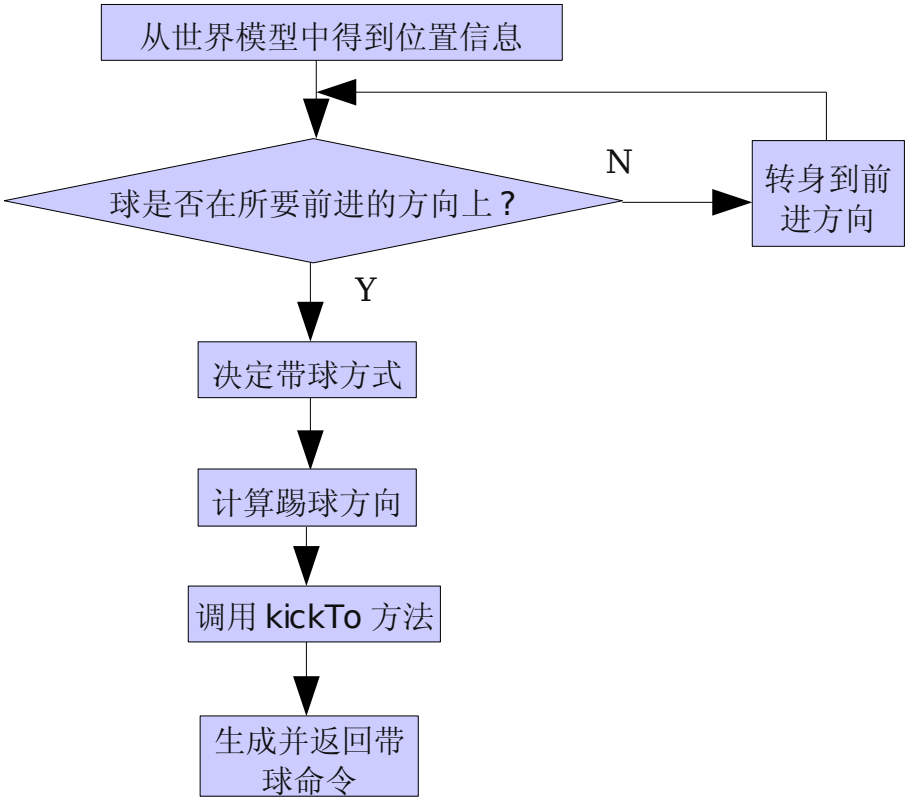


图 5.1 dribble 方法执行流程图

5.2.3 interceptClose 断球方法的实现

interceptClose 方法使球员能够在球接近自己时断球。该方法的目的是寻找到一个组合使球员能在两个时间周期内移动到球能够被自己踢到的范围内为了达到这个目的，需要调用世界模型中的预测方法来预测球在未来一到两个周期内的位置。接着判断所有 turn 方法和 dash 方法的逻辑组合中是否存在至少一种组合能够使球员在两个时间周期内进入可以踢到球的范围内。如果不存在的话，则放弃使用断球的方法，退出。如果存在这种组合，则执行该动作组合。

interceptClose 方法执行流程见图 5.2

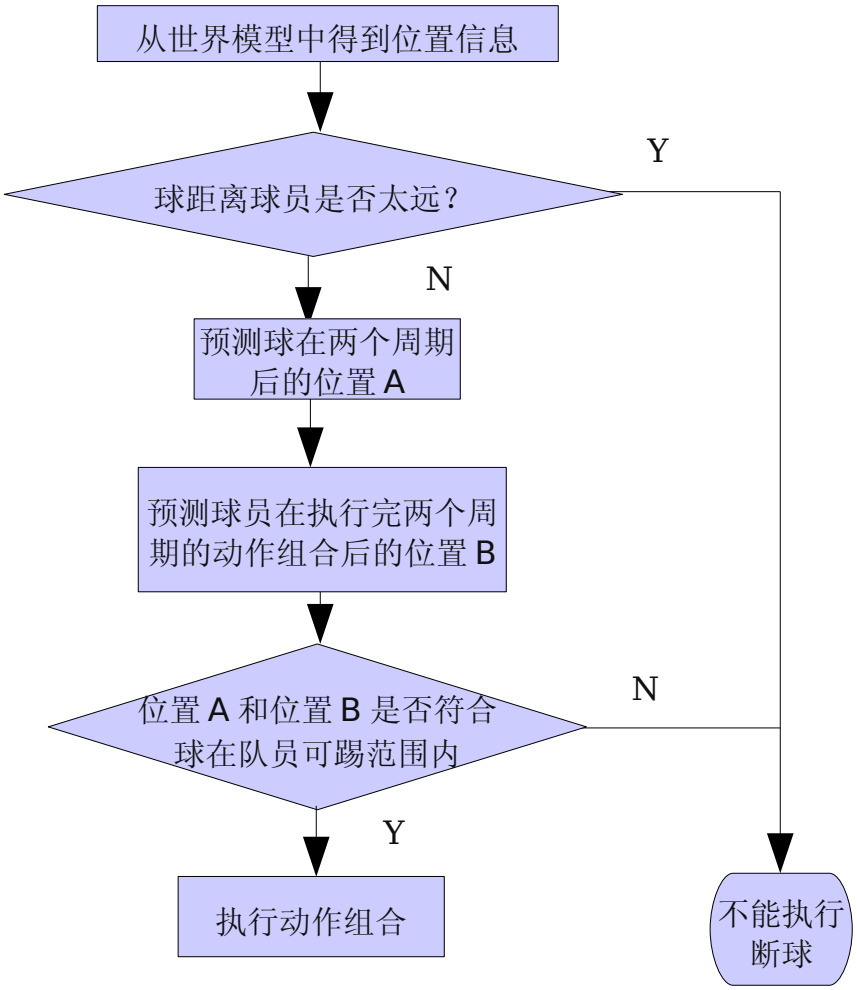


图 5.2 interceptClose 方法流程图

5.2.4 directPass 传球方法的实现

directPass 方法使球员能够将球直接传给队友。该方法使用两个参数，pos 和 passType。pos 代表球所传向的位置，passType 代表传球的方式。

passType 包括两种，PASS_NORMAL 和 PASS_FAST。它们的不同之处在于传出的球的末速度不同，使用前者要比使用后者产生的末速度小。

directPass 方法实际上就是给定了合适的位置和速度参数的 kickTo 方法，目的是提高队友成功接到球的概率。

本方法的流程如图 5.3

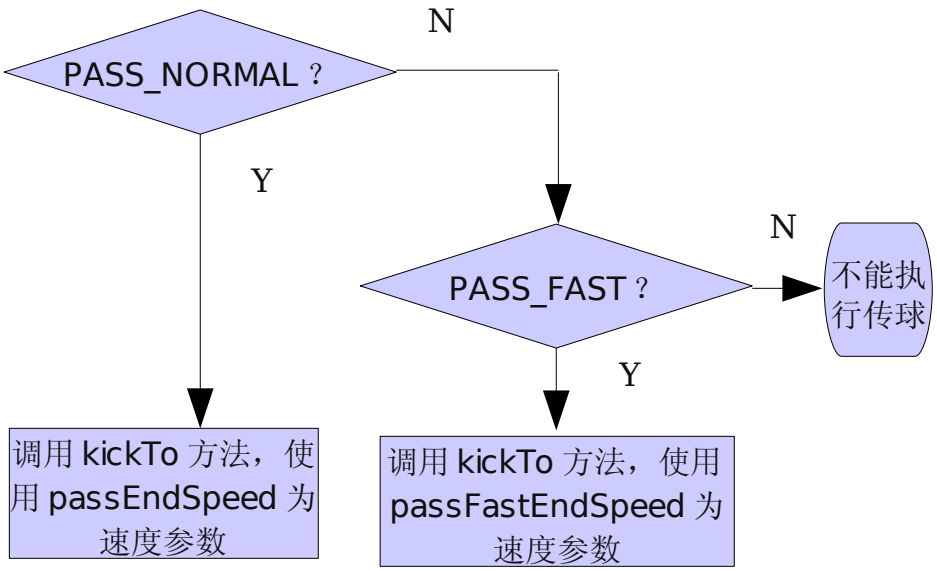


图 5.3 directPass 方法流程图

directPass()方法的源代码如下

```
SoccerCommand BasicPlayer::directPass( VecPosition pos, PassT passType)
{
   if( passType == PASS_NORMAL )
```

```
    return kickTo( pos, PS->getPassEndSpeed() );
  else if( passType == PASS_FAST )
    return kickTo( pos, PS->getFastPassEndSpeed() );
  else
    return SoccerCommand( CMD_ILLEGAL );
}
```

6 球员间配合的设计

6.1 配合的实现方法

任何一个单独的球员不可能完成整个比赛，因此球员间的配合能力是比赛成功与否的关键。配合能力是指每个球员均可以与其它队友进行配合，比如可以将球有目的的传给某一个队友，可以接到队友的传球等等，而不是无目的的断球，踢球和传球。配合的能力则由配合策略实现。

配合策略是由 Player 类中的两个方法，toxic 方法和 toxic_goalie 方法来实现。toxic 方法实现了普通球员的策略，而 toxic_goalie 方法实现了守门员的策略。

6.2 普通球员策略的设计

在本设计中，普通球员间的配合采用基于球员角色分配的配合策略。将各个队员进行任务分工，主要是将普通球员分成两大类：攻击型和配合型。攻击型球员主要任务是接到球之后，直接进攻对方球门。配合型球员的主要任务是将球传送给攻击型球员。

6.2.1 攻击型球员策略

攻击型球员的策略可以概述如下：当得到球之后，计算自己与对方守门员的距离，如果距离小于等于 15，射门；如果距离在 15 和 25 之间，带球向球门前进；如果距离大于等于 25，将球传给另一个攻击型队员。每个攻击型球员的详细策略流程如图 6.1：

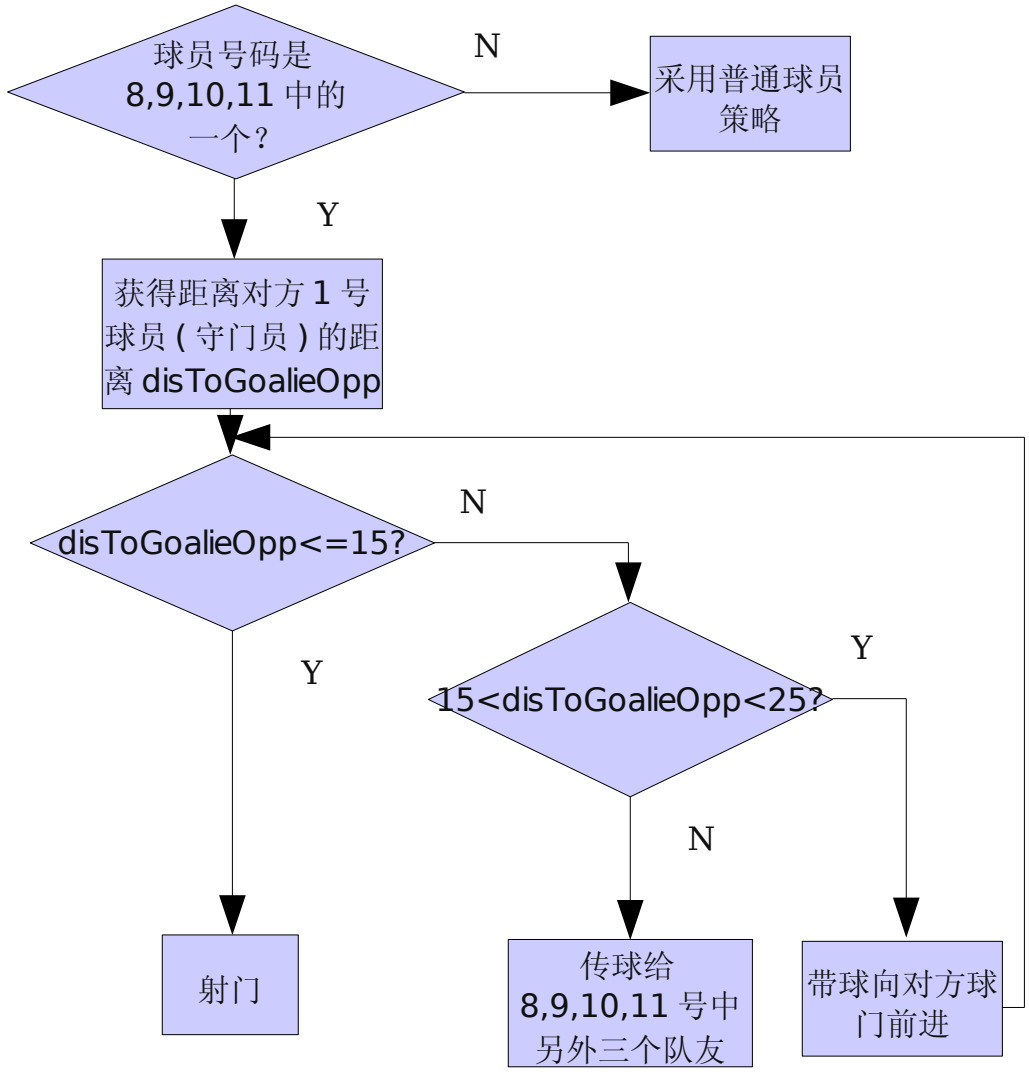


图 6.1 攻击型球员策略流程图

这里射门动作本质上是一个特殊化了的 kickTo 方法，在踢球时使用最大的力量，并且踢球的方向是对方球门的角落。带球采用 dribble 方法，带球方式为快速带球，带球的方向是对方球门。传球采用 directPass 方法，传球方式为快速传球。

6.2.2 配合型球员策略

配合型球员策略的目的就是成功的将球传给攻击型球员。在本设计中 2 号至 7 号球员均为配合型球员。在采用 7.2.1 中的 443 阵形时，其中 2、3、5 号球员的位置在后场，6、4、7 号球员的位置在中场。根据后场球员与前场球员之间配合方式的不同，采用了两种方案来实现配合型球员的策略。

第一种方案称为双路方案，如图 6.2，即 2、3 号队员向 7 号队员传球，4、5 号球员向 6 号球员传球。本方案采取左右两路的方式将球从后场和中场由配合型球员向攻击型球员传递。

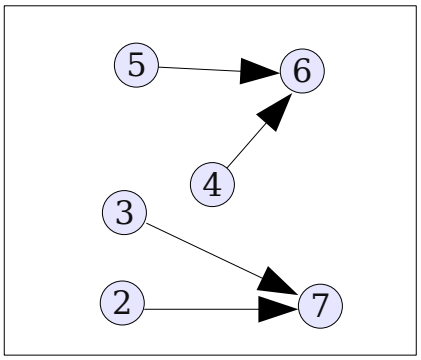


图 6.2 双路方案传球路线

第二中方案称为中路方案，如图 6.3，即 2、3、5 号球员向 4 号球员传球，4 号球员将球传给 6 号或 7 号球员。本方案采用单独中路传球的方式将球从后场和中场由配合型球员传给攻击型球员。

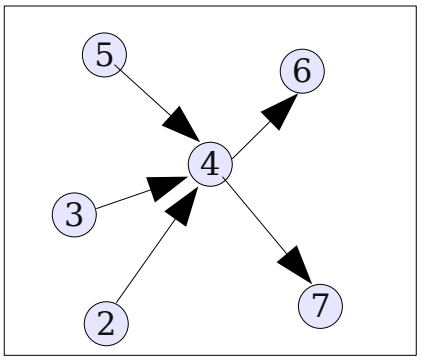


图 6.3 中路方案传球路线

以上两种方案的使用效果在 9.3 中进行了评估。

这里传球使用的是 directPass 方法，采用的传球方式为快速传球。

6.3 守门员策略的设计

根据比赛的不同状态设计不同的守门员策略。这里介绍正常比赛中的扑球策略和开球门球时的开球策略的实现。

6.3.1 守门员扑球策略

守门员最重要的一个技能就是可以执行 catchBall 的动作，即抓住球。因此守门员策略中最重要的部分就是如何应用 catchBall 方法，使能够成功扑住对方的射门，而不是简单的将球踢出。

守门员的扑球策略可以概述如下：检测当前比赛状态，只有在正常比赛中，即非开场前，也非球门球，才能实施扑球策略。定义一个矩形来代表自己所守的球门禁区。守门员的活动范围仅限于这个矩形当中。当有球在这个区域内时，定义一条直线，该直线贯穿球门中心和球的中心，守门员移动到这条直线上一个离自己最近的位置，并沿着这条线奔向球，以拦截球。当球在自己的抓球范围内时，调用 catchBall 方法将球扑住。如图 6.4

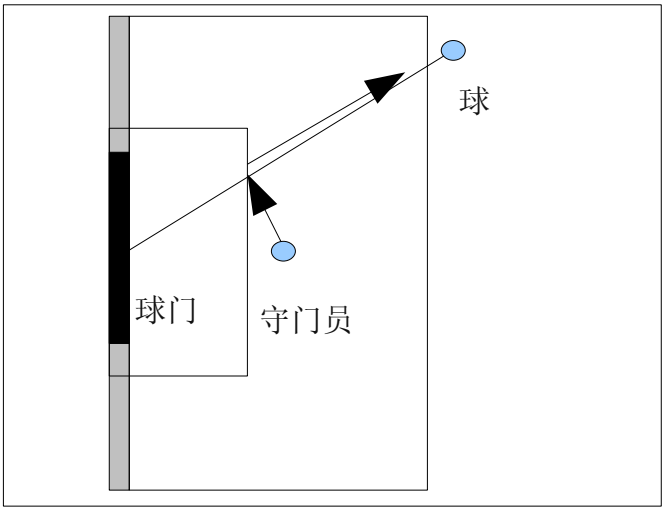


图 6.4 守门员扑球路线

扑球策略详细流程见图 6.5

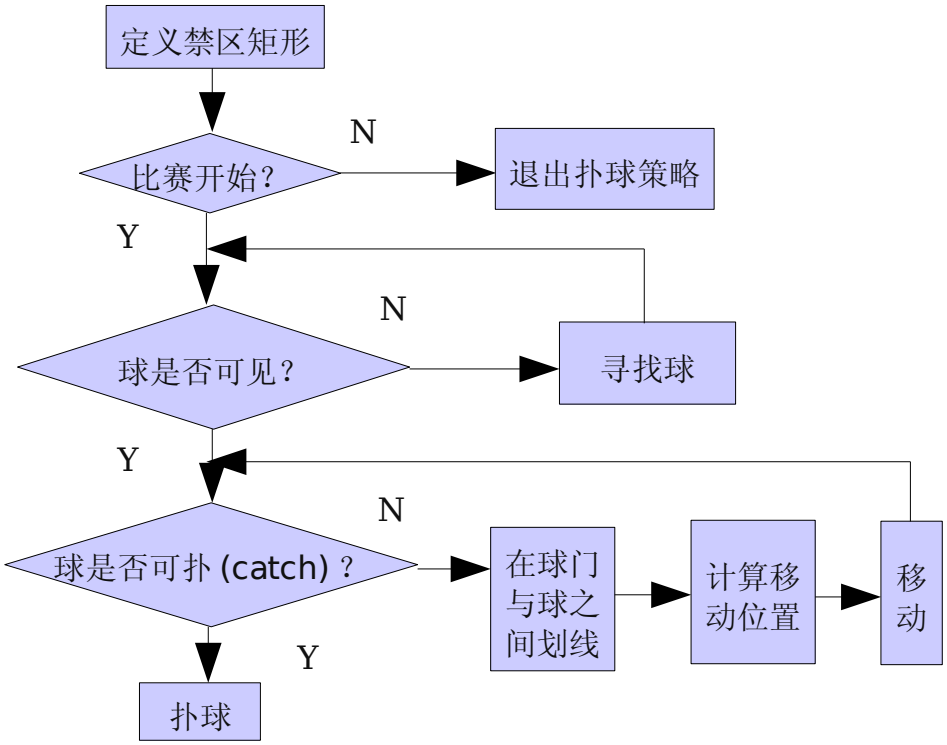


图 6.5 守门员扑球策略

6.3.2 守门员开球门球策略

守门员将球扑住之后开球门球的策略概述如下：定义一个圆形区域，寻找禁区前沿在此圆形区域内对方球员最少的区域，然后移动到这个区域，将球使用 kickTo 方法开出。

7 阵形的设计

7.1 阵形设计的原理

球队的阵形即指球队所有队员静止时在场上的站位。由于守门员位置始终是在禁区内部，所以在不同的阵形中守门员的位置是相同的。其它 10 名队员根据不同的球队策略将有不同的位置，进而组成不同的阵形，比如 334 阵

形，343 阵形等。

本设计中阵形数据并未编译进球队程序，而是作为外部文件 formations.conf 存储。这样能够方便的进行阵形编辑和队员位置微调，而不用重新编译球队程序。球队程序启动的时候将存储阵形数据的外部文件读入根据文件中的数据进行不同队员位置参数的设置。

阵形文件数据的基本结构如下：

```
1
-50.0 -16.0 -17.0 -17.0 -16.0  -8.0  -2.0  -2.0  -2.0  -1.0  -1.0
  0.0  16.0   5.0  -5.0 -16.0   0.0  10.0  -2.0   0.0  22.0 -22.0
```

第一排的数字代表阵形的序号，因为一个阵形数据文件中包含不只一个阵形。第二排 11 个数字代表 11 个队员在场上对应的 X 坐标，第三排 11 个数字代表 11 个队员在场上对应的 Y 坐标。

Formations 类的 readFormations 方法根据阵形文件中的数据设定队员的坐标。readFormations 方法利用 Formations 类中的 fs 方法将文件读入，并利用 setXPosHome 方法和 setYPosHome 方法设置各个球员在场上的位置。

7.2 阵形在比赛中的应用

7.2.1 阵形的种类

根据进攻和防守的需要，并参考 Trilearn 球队的阵形，一共设计了 5 常见的阵形，详细见表 7.1

表 7.1 常见阵形列表

阵形名	阵形种类	特点
FT_433_offensive	进攻阵形	3 名前锋
FT_334_offensive	进攻阵形	4 名前锋
FT_defensive(442)	防守阵形	4 名后卫
FT_open_defensive(442)	防守阵形	4 名后卫，加强一定进攻
FT_343_attacking	进攻阵形	3 名前锋，加强中场

7.2.2 阵形的转换

首先通过定义一个 FormationT 的枚举类型，包含以上 5 种阵形。通过 Formations 的 setFormation 方法可以设定球队当前的阵形。setFormation 方法使用 FormationT 为参数，因此可以方便的调用设定好的阵形数据。

通过下面的代码段举例说明了如何使球队在比赛时间 1000 之后将阵形变为 442 阵形，而在时间 1000 之前采用 334 阵形。

```
if (WM->getCurrentTime()>=1000)
{
    formations->setFormation( FT_DEFENSIVE );//442 阵形
}
else
    formations->setFormation( FT_334_OFFENSIVE );//334 阵形
```

8 场上教练的设计

8.1 场上教练介绍

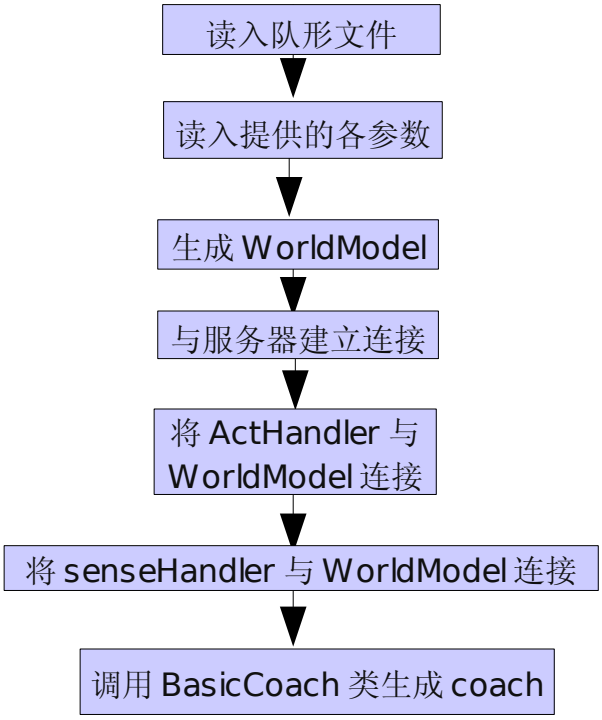
场上教练又被称为在线教练。场上教练作用是向球员提供帮助，给队员策略指示。场上教练得到的场上信息没有噪音，并且服务器对其实时性要求不是太高，因此场上教练通常起的作用是进行球队策略的选择。同时为了避免引入场上教练而使比赛背离分布式控制的原则，导致比赛变成集中式控制教练与服务器通信的能力及可供选择的命令种类受到了限制。

8.2 简单的场上教练的设计

本设计实现了一个简单的场上教练，实现了替换 Heterogeneous 球员类型的功能。Heterogeneous 球员即异类球员，是在 rcssserver 版本 7 以后引入的。每个球员在比赛开始的时候被服务器随机初始化为 7 中不同类型球员中的某一种类型。不同类型的球员根据 player.conf 文件中的参数值而拥有不同的能力。场上教练在比赛开始之前可以无限次更改队员的类型，但是在比赛开始后只能改变 3 次。

详细的球员类型的应用比较复杂，本设计仅仅演示了如何使用场上教练去改变球员的类型，这些改变目前还并不是策略的一部分，即球员类型的改变影响不到目前策略实施的效果。

场上教练的设计与 agent 的设计很类似，并且与 agent 使用了大量相同的类。场上教练程序启动的流程图如图 8.1



9 比赛测试结果分析与评估

9.1 三种带球方式的比较

5.5.2 中设计了三种带球的方式，DRIBBLE FAST, DRIBBLE SLOW, 和 DRIBBLE WITH BALL。通过对多场比赛的统计，在抢断队员使用同样的抢断策略：即以最大的速度跑向球并使用 interceptClose 方法断球的情况下，采用以上三种带球方式的 9 号球员带球成功率的统计结果表 9.1：

表 9.1 三种带球方式成功率的统计表

带球方式	被断球次数/带球	带球成功率
DRIBBLE FAST	15/30	50%
DRIBBLE SLOW	20/35	42.9%
DRIBBLE WITH BALL	30/35	14.3%

其中 DRIBBLE FAST 带球方式的成功率相比之下最高，因为相对高速度的带球虽然球离身体比其它两种方式较远，但对付正面的拦截却非常有效；而 DRIBBLE WITH BALL 虽然能够将球控制在离自己很近的范围内，但对正面拦截起不了太大的作用。

基于上述统计，并且根据比赛中正面拦截出现的几率比较高的事实，总体策略中使用了 DRIBBLE FAST 的带球方式。

9.2 使用 kickTo 方法传球和使用 directPass 方法传球的比较

传球可以使用 kickTo 方法来实现，这是最基本的方式。但是由于 kickTo 方法中很难准确的确定队友的位置和传球速度，因此影响了传球的成功率。在 5.2.4 中设计了特殊化的 kickTo 方法即 directPass 方法，目的即是为了得到更高的传球成功率。

使用 kickTo 方法的传球策略和使用 directPass 方法代替 kickTo 方式后的策略下总的传球成功率比较见表 9.2。

表 9.2 的结果是在统计了半场比赛中我方的传球数据后得出的。

表 9.2 使用 kickTo 方法与使用 directPass 方法两种传球方式的成功率比较

传球方式	传球成功次数/传球总次数	传球成功率
kickTo 方法	1226/2150	57.0%
directPass 方法	1903/2203	86.4%

由表 9.2 可以看出，特殊化了的 kickTo 方法即 directPass 方法能够大幅度提高传球的成功率。因此总体策略中的传球策略采用了 directPass 方法。

9.3 双路方案与中路方案的比较

最终设计好的球队 Ncut_Alf_Toxic 的配合型球员在采用双路方案与中路方案的情况下分别与 TrilearnBase2003 球队进行比赛，TrilearnBase2003 球队球员间配合的策略固定如下：任何球员在得球后踢向敌方球门，并且离球最近的球员断球。双方球队均采用 334 阵形。采用以上两种方案的多场比赛结果统计如表 9.3

表 9.3 在双路方案与中路方案下 Ncut_Alf_Toxic 对 TrilearnBase2003 比赛结果

比赛场次	双路方案下的比分	中路方案下的比分
1	3:0	12:4
2	2:0	7:2
3	4:1	9:2
4	3:1	5:2
5	4:1	10:1

注：TrilearnBase2003 球队是由 Uva-Trilearn 球队公开的源代码所编译成的球队，也是本设计所参考的球队。

由表 9.3 的统计结果可知，我方球队在配合型球员使用双路方案和使用中路方案的情况下均可以战胜对方球队。在双路方案下 5 场比赛我方进球总数为 16 个，失球总数为 3 个，而在中路方案下 5 场比赛我方进球总数达 53 个，失球总数上升为 11 个。因此，中路方案下的进攻比双路方案下的进攻的效果提高了许多，不过中路方案下的防守比双路方案下的防守的成功率要低，因此采用中路方式时我方进球数和失球数均有大幅度上升。

9.4 比赛的总体效果

将我方最终设计好的球队 Ncut_Alf_Toxic 与 Trilearn2003Base 球队进行比赛，我方球队采用 6.2 中设计的策略，其中配合型球员策略采用中路方案，阵形为 334 和 343；TrilearnBase2003 球队间配合策略同 9.3 中 TrilearnBase2003 球队使用的策略，阵形采取固定的 334 进攻阵形。多场比赛的统计结果如表 9.4

表 9.4 Ncut_Alf_Toxic 对 TrilearnBase2003 比赛结果

我方阵形/敌方阵形	Ncut_Alf_Toxic	Trilearn2003Base
334/334	12	4
334/334	7	2
334/334	9	2
334/334	5	2
334/334	10	1
343/334	2	3
343/334	3	2
343/334	0	0
343/334	1	1
343/334	1	2

由表 9.4 可以看出我方当前使用的总体策略在 334 阵形下，5 场比赛进球达 53 个，失球 11 个，而在 343 阵形下进球只有 7 个，失球却达 8 个。因此 6.2 中的策略在 334 的阵形下能够达到最好的效果，而 343 的阵形下，策略得不到完全的发挥。需要设计新的配合方式。

结论

本论文完成了一个完整的可以进行正常 RoboCup 2D 仿真组比赛的足球队。完成的具体工作及成果如下：

(1) 设计了球员的多种场上技能，比如带球、传球、断球等。

(2) 提出并实现了球员间的配合策略，守门员的防守策略，并通过实际比赛证明了其有效性。

(3) 实现了球队的阵形功能，比如 334 阵形和 343 阵形等，并实现了比赛中阵形的转换。

(4) 实现了一个基本的可以起简单自主在线指挥作用的场上教练程序。

(5) 对不同的带球方法、传球方法和配合策略的效果进行了评估，并通过实际比赛，选出最优的策略供球队使用。

由于知识水平以及时间的限制，一些新的策略和想法没有得到实现。目前的球队策略需要在以下几个方面改进：

(1) 深入研究 Heterogeneous 球员类型，进而能够使用场上教练在比赛时对球员类型的分配进行优化。

(2) 对不同的阵形编写相应最有效的配合策略，并实现在队形变化时配合策略随之改变。

(3) 引入更多的人工智能的知识，比如神经网络，再励学习等方法设计球队高层策略，使球队更加智能。

致谢

感谢 Uva-Trilearn 球队提供的 TrilearnBase2003 源代码，本设计基于其设计思想并使用了世界模型、通信部分、球员部分技能的源代码并参考了其队形数据。

感谢李小坚教授的辛勤指点和教导，使我有机会能够对 RoboCup 机器人比赛进行更多的深入接触，并有幸能够进行整个 2D 仿真球队的设计。

感谢我的父母和我的女朋友聂胜男，感谢你们在我最忙的半年中始终对我的毫无保留的支持、理解和关心。

参考文献

1 Mao Chen,Klaus Dorer 等， Users Manual-RoboCup Soccer Server for Soccer Server 7.07 and later，http://sserver.sourceforge.net/，2003

2 M. Riedmiller，A. Merke，D. Meier，A. Hoffmann，A. Sinner，O. Thate，R. Ehrmann，Karlsruhe Brainstormers – A Reinforcement Learning approach to robotic soccer，Lecture Notes in Computer Science，2001，2019，367-372

3 郭叶军，机器人足球仿真比赛中多智能体系统的构建，http://www.nlict.zju.edu.cn/nlictrobocup/zjubase_pub/master.pdf，2004.3

4 Stanley B.Lippman，C++ Primer，英文第三版，北京，人民邮电出版社，2005.9

5 官俊、罗迪君、梁达明，ZJUBase 体系架构：实现一个合作团队，http://www.nlict.zju.edu.cn/nlictrobocup/zjubase_pub/gz.pdf，2004.10

附录 1：源程序清单列表

/src

ActHandler.cpp GenericValues.cpp Parse.cpp ServerSettings.h
ActHandler.h GenericValues.h Parse.h SoccerTypes.cpp
BasicCoach.cpp Geometry.cpp Player.cpp SoccerTypes.h
BasicCoach.h Geometry.h Player.h WorldModel.cpp
BasicPlayer.cpp Logger.cpp PlayerSettings.cpp WorldModel.h
BasicPlayer.h Logger.h PlayerSettings.h WorldModelHighLevel.cpp
Connection.cpp mainCoach.cpp
PlayerTeams.cpp WorldModelPredict.cpp
Connection.h main.cpp SenseHandler.cpp WorldModelUpdate.cpp
Formations.cpp Objects.cpp SenseHandler.h
Formations.h Objects.h ServerSettings.cpp
Makefile
formations.conf
player.conf

Makefile

start.sh

更多关于源程序的信息，以及源程序的下载请访问本毕业设计的网站 http://zjfroot.googlepages.com/robocup

附录 2：实现寻找球的 searchBall()方法的源代码

```
SoccerCommand BasicPlayer::searchBall()
{
   static Time   timeLastSearch;
   static SoccerCommand soc;
   static int    iSign       = 1;
   VecPosition   posBall =WM->getBallPos();
   VecPosition   posAgent=WM->getAgentGlobalPosition();
  AngDeg        angBall =(posBall-WM->getAgentGlobalPosition()).getDirection();
   AngDeg        angBody =WM->getAgentGlobalBodyAngle();
   if( WM->getCurrentTime().getTime() == timeLastSearch.getTime()  )
      return soc;
   if( WM->getCurrentTime() - timeLastSearch > 3 )
      iSign = ( isAngInInterval( angBall, angBody,
                        VecPosition::normalizeAngle(angBody+180) ) )
          ? 1  : -1  ;
   soc = turnBodyToPoint( posAgent + VecPosition(1.0,
VecPosition::normalizeAngle(angBody+60*iSign), POLAR ) );//转身一个角度
   Log.log( 556, "search ball: turn to %f s %d t(%d %d) %f", angBall, iSign,
     WM->getCurrentTime().getTime(), timeLastSearch.getTime(),soc.dAngle );
   timeLastSearch = WM->getCurrentTime();
   return soc;
}
```

附录 3: 实现带球的 dribble()方法的源代码

```
SoccerCommand BasicPlayer::dribble( AngDeg ang, DribbleT dribbleT )
{
   double       dLength;
   AngDeg       angBody  = WM->getAgentGlobalBodyAngle();
   VecPosition   posAgent = WM->getAgentGlobalPosition();
   SoccerCommand soc;
   //若球不在自己带球的方向，转身至该方向
   AngDeg angDiff = VecPosition::normalizeAngle( ang - angBody );
   if( fabs( angDiff ) > PS->getDribbleAngThr() )
      return turnWithBallTo( ang, PS->getTurnWithBallAngThr(),
                                          PS->getTurnWithBallFreezeThr() );
   switch( dribbleT )
   {
      case DRIBBLE_WITHBALL:
         dLength = 4.0;
         break;
      case DRIBBLE_SLOW:
         dLength = 5.0;
         break;
      case DRIBBLE_FAST:
         dLength = 10.0;
```

```
            break;
        default:
            dLength = 0.0;
            break;
    }

    // 决定带球位置
    VecPosition posDribble = posAgent + VecPosition( dLength, angBody, POLAR )
    soc = kickTo( posDribble, 0.5 );
    return soc;
}
```

附录 4：实现断球的 interceptClose()方法的源代码

```
SoccerCommand BasicPlayer::interceptClose( )
{
   FeatureT feature_type = FEATURE_INTERCEPT_CLOSE;
   if( WM->isFeatureRelevant( feature_type ) )
      return WM->getFeature( feature_type ).getCommand();

            SoccerCommand soc, socDash1, socFinal, socCollide,
socTurn(CMD_ILLEGAL);
   double dPower, dDist;
   AngDeg ang, ang2;
   VecPosition s1, s2;
   bool bReady = false;

   dDist = 3*SS->getPlayerSpeedMax()
            + (1.0 + SS->gSoccerCommand BasicPlayer::interceptClose( )
{
   FeatureT feature_type = FEATURE_INTERCEPT_CLOSE;
   if( WM->isFeatureRelevant( feature_type ) )
      return WM->getFeature( feature_type ).getCommand();

            SoccerCommand soc, socDash1, socFinal, socCollide,
```

```
socTurn(CMD_ILLEGAL);
   double       dPower, dDist;
   AngDeg       ang,    ang2;
   VecPosition  s1,     s2;
   bool         bReady = false;

   //首先判断是否距离球太远
   dDist = 3*SS->getPlayerSpeedMax()
               + (1.0 + SS->getBallDecay())*SS->getBallSpeedMax()
               + SS->getMaximalKickDist();
   if( WM->getRelativeDistance( OBJECT_BALL ) > dDist )
   {
      bReady = true;
      socFinal = SoccerCommand( CMD_ILLEGAL ); //放弃使用拦截动作
   }

   socCollide = collideWithBall( );
   // 初始化世界模型中的位置信息
   VecPosition  posAgent = WM->getAgentGlobalPosition( );
   VecPosition  posPred  = WM->predictAgentPos( 1, 0 ), posDash1;
   VecPosition  posBall  = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
   VecPosition  velMe    = WM->getAgentGlobalVelocity( );
```

```
Stamina      sta     = WM->getAgentStamina( );
AngDeg       angBody = WM->getAgentGlobalBodyAngle( ), angTurn, angNeck=0;
double       dDesBody = 0.0;

if( posAgent.getX() > PENALTY_X - 5.0 )
   dDesBody = (WM->getPosOpponentGoal()-posAgent).getDirection();

double dDistOpp;
ObjectT objOpp = WM->getClosestInSetTo( OBJECT_SET_OPPONENTS,
    WM->getAgentObjectType(), &dDistOpp, PS->getPlayerConfThr() );
                angTurn =VecPosition::normalizeAngle(dDesBody-WM->getAgentGlobalBodyAngle());

// 判断不做 dash 动作时距离球的位置
posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
VecPosition posPred1 = WM->predictAgentPos( 1, 0 );
double dDist1 = posPred1.getDistanceTo( posBall );
posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 2 );
VecPosition posPred2 = WM->predictAgentPos( 2, 0 );
double dDist2 = posPred2.getDistanceTo( posBall );
posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 3 );
```

```
    VecPosition posPred3 = WM->predictAgentPos( 3, 0 );
    double dDist3 = posPred3.getDistanceTo( posBall );
    Log.log( 508, "dist 1: %f, 2: %f 3: %f, 0.6: %f", dDist1, dDist2, dDist3,
                                                    0.7*SS->getMaximalKickDist() )
;

    AngDeg angThreshold = 25;
    bool   bOppClose = ( objOpp != OBJECT_ILLEGAL && dDistOpp < 3.0 )  ;
    posAgent  = WM->getAgentGlobalPosition( );
    posBall   = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
    angBody   = WM->getAgentGlobalBodyAngle();
    velMe     = WM->getAgentGlobalVelocity( );
    sta       = WM->getAgentStamina( );
    Line line = Line::makeLineFromPositionAndAngle(posPred1,angBody);
    dDist     = SS->getPlayerSize()+SS->getBallSize()+SS->getKickableMargin()/6;
    int  iSol = line.getCircleIntersectionPoints(
                                                    Circle(posBall,dDist), &s1,
&s2);
    if (iSol > 0)                          //如果有解
    {
       if (iSol == 2)
       {
```

```
        ang = VecPosition::normalizeAngle((posBall - s1).getDirection() -angBody);
        ang2= VecPosition::normalizeAngle((posBall - s2).getDirection() -angBody);
//    if ( fabs(ang2) < 90)
        if( s2.getX() > s1.getX() ) // 尽最大可能前进
            s1 = s2;
      }

//判断一次 dash 是否满足条件
                    dPower = WM->getPowerForDash(s1-posAgent, angBody,
velMe,sta.getEffort() );
      posDash1 = WM->predictAgentPos( 1, (int)dPower);
      if ( posDash1.getDistanceTo( posBall ) < 0.95*SS->getMaximalKickDist() )
      {
         Log.log( 508, "dash 1x possible at s1" );
         socDash1 = SoccerCommand( CMD_DASH, dPower );
      }
      else
      {
                          dPower=WM->getPowerForDash(s2-posAgent, angBody,
velMe,sta.getEffort() );
         posDash1 = WM->predictAgentPos( 1, (int)dPower);
                      if ( posDash1.getDistanceTo( posBall ) < 0.95*SS-
```

```
>getMaximalKickDist() )
        {
          Log.log( 508, "dash 1x possible at s2" );
          socDash1 = SoccerCommand( CMD_DASH, dPower );
        }
      }
    }
    if( socDash1.commandType == CMD_ILLEGAL )
    {
      soc = dashToPoint( posBall );
      WM->predictAgentStateAfterCommand(soc,&posDash1,&velMe,
                                        &angBody,&ang,&sta );
      if ( posDash1.getDistanceTo( posBall ) < 0.95*SS->getMaximalKickDist() )
      {
        Log.log( 508, "dash 1x possible (special)" );
        socDash1 = soc;
      }
    }
//寻找可用动作组合，并执行
    if( bReady != true )
    {
      if( bOppClose && ! socDash1.isIllegal() )
```

```
    {
      Log.log( 508, "do dash 1x, opponent close" );
      WM->logCircle( 508, posDash1, SS->getMaximalKickDist(), true );
      socFinal = socDash1;
    }
    else
    {
      soc = turnBodyToPoint( posPred1 + VecPosition(1,dDesBody, POLAR), 1 )
;
      WM->predictAgentStateAfterCommand(soc, &posPred, &velMe,
                                                &angBody, &ang, &sta);
      posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
      if( posPred.getDistanceTo( posBall ) < 0.8*SS->getMaximalKickDist() )
      {
        socTurn = soc;
        ang     = VecPosition::normalizeAngle(dDesBody-angBody);
        if( fabs(ang) < angThreshold )
        {
          socFinal = soc;
          Log.log( 508, "turn 1x, dist %f, angle %f, opp %f",
              dDist1, angTurn, dDistOpp );
          WM->logCircle( 508, posPred1, SS->getMaximalKickDist(), true );
```

```
    }

}
if( socFinal.isIllegal() )
{
    ang     = VecPosition::normalizeAngle(dDesBody-angBody);
    WM->predictStateAfterTurn(
                        WM->getAngleForTurn(ang,velMe.getMagnitude()),
                        &posPred, &velMe, &angBody, &angNeck,
                        WM->getAgentObjectType(),
                        &sta            );
    posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 2 );
    if( posPred.getDistanceTo( posBall ) < 0.8*SS->getMaximalKickDist() )
    {
        socTurn = soc;
        ang     = VecPosition::normalizeAngle(dDesBody-angBody);
        if( fabs(ang) < angThreshold )
        {
            Log.log( 508, "turn 2x, dist %f, angle %f, opp %f",
                        dDist2, angTurn, dDistOpp );
            WM->logCircle( 508, posPred2, SS->getMaximalKickDist(), true );
```

```
            socFinal = soc;
          }
        }
      }
      if( socFinal.isIllegal() && ! socCollide.isIllegal() &&
          fabs( angTurn ) > angThreshold )
      {
        Log.log( 508, "collide with ball on purpose" );
        posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 1 );
        WM->logCircle( 508, posBall, SS->getMaximalKickDist(), true );
        socFinal = socCollide;
      }
      if( socFinal.isIllegal() && fabs( angTurn ) > angThreshold )
      {
        posBall = WM->predictPosAfterNrCycles( OBJECT_BALL, 2 );
        soc = dashToPoint( posAgent );
        WM->predictAgentStateAfterCommand(soc,
                                          &posPred,&velMe,&angBody,
&ang,&sta );
        if( posPred.getDistanceTo( posBall ) < 0.8*SS->getMaximalKickDist() )
        {
          Log.log( 508, "dash 1x (stop), turn 1x, dist %f, angle %f, opp %f",
```

```
                dDist2, angTurn, dDistOpp );
            WM->logCircle( 508, posPred, SS->getMaximalKickDist(), true );
            socFinal = soc;
        }
    }
    if( socFinal.isIllegal()  && ! socTurn.isIllegal() )
    {
        Log.log( 508, "can do turn" );
        WM->logCircle( 508, posPred1, SS->getMaximalKickDist(), true );
        socFinal = socTurn;
    }
    if( socFinal.isIllegal()  && ! socDash1.isIllegal() )
    {
        Log.log( 508, "do dash 1x" );
        WM->logCircle( 508, posDash1, SS->getMaximalKickDist(), true );
        socFinal = socDash1;
    }
  }

}

// 三个周期内如果仍然断不下球放弃断球
```

```
    WM->setFeature( feature_type,
                    Feature( WM->getTimeLastSeeMessage(),
                             WM->getTimeLastSenseMessage(),
                             WM->getTimeLastHearMessage(),
                             OBJECT_ILLEGAL,
                             -1,
                             socFinal ) );
    return socFinal;
}
```

附录 5：实现普通球员策略的 toxic()方法的源代码

```
SoccerCommand Player::toxic(  )
{
   SoccerCommand soc(CMD_ILLEGAL);
   VecPosition   posAgent = WM->getAgentGlobalPosition();//得到球员自己的位置
   VecPosition   posBall  = WM->getBallPos();//获得球的位置
   int          iTmp;

   if( WM->isBeforeKickOff( ) )//中场开球前球员位置处理
   {
      if( WM->isKickOffUs( ) && WM->getPlayerNumber() == 8 ) // 8 号开球
      {
         if( WM->isBallKickable() )
         {

         VecPosition posGoal( PITCH_LENGTH/2.0,
                                        (-1 + 2*(WM->getCurrentCycle()%2)) *
                                        0.4 * SS->getGoalWidth() );//定义踢球方向
            soc = kickTo( posGoal, SS->getBallSpeedMax() ); // 使用最大力气踢球
            Log.log( 100, "take kick off" );
         }
         else
```

```
        {
            soc = intercept( false );
            Log.log( 100, "move to ball to take kick-off" );
        }
        ACT->putCommandInQueue( soc );
        ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
        return soc;
    }
    if( formations->getFormation() != FT_INITIAL || // 不在 FT_INITAL 模实现
下
        posAgent.getDistanceTo( WM->getStrategicPosition() ) > 2.0 )
    {
        formations->setFormation( FT_INITIAL );      // 改变阵形到 kickoff 阵形
                        ACT->putCommandInQueue(    soc=teleportToPos(    WM-
>getStrategicPosition() ));
    }
    else                                // 转身至中心
    {
        ACT->putCommandInQueue( soc=turnBodyToPoint( VecPosition( 0, 0 ),
0 ) );
        ACT->putCommandInQueue( alignNeckWithBody( ) );
    }
  }
  else
  {
```

```
//      if (WM->getCurrentTime()>=300)
//      {
//          formations->setFormation( FT_DEFENSIVE );//改变为 442 阵形
//      }
//      else
            formations->setFormation( FT_334_OFFENSIVE );
//          formations->setFormation( FT_343_ATTACKING );
        soc.commandType = CMD_ILLEGAL;
        double disToGoalieOpp;
        if( WM->getConfidence( OBJECT_BALL ) < PS->getBallConfThr() )
        {
          ACT->putCommandInQueue( soc = searchBall() );   // 如果球的位置未知
          ACT->putCommandInQueue( alignNeckWithBody( ) ); // 寻找球
        }
        else if( WM->isBallKickable())                  // 如果球可踢
        {
            AngDeg angleToChose=WM->getRelAngleOpponentGoal();//获得对方
                                                          //守门员距离自己的距离。
            if(WM->getPlayerNumber()==9)
            {
                disToGoalieOpp=WM-
                                >getRelativeDistance(OBJECT_OPPONENT_1);
                if(disToGoalieOpp<=15.0)
                {
                    VecPosition posGoal( PITCH_LENGTH/2.0,
                            (-1 + 2*(WM->getCurrentCycle()%2)) * 0.4 * SS-
```

```
                                                    >getGoalWidth() );
            soc = kickTo( posGoal, SS->getBallSpeedMax() );
        }
        else if(disToGoalieOpp>15.0&&disToGoalieOpp<=25.0)
        {
            soc=dribble(angleToChose,DRIBBLE_FAST);
        }
        else
        {   VecPosition   posNr10=WM-
                          >getGlobalPosition(OBJECT_TEAMMATE_10);
            soc=directPass(posNr10,PASS_FAST);
        }
}//9 号攻击型队员的策略实现
else
if(WM->getPlayerNumber()==7)
{
    VecPosition   posNr9=WM-
                    >getGlobalPosition(OBJECT_TEAMMATE_9);
    soc=directPass(posNr9,PASS_FAST);
}//7 号配合型队员的策略实现
else
if(WM->getPlayerNumber()==8)
{
    disToGoalieOpp=WM-
                    >getRelativeDistance(OBJECT_OPPONENT_1);
    if(disToGoalieOpp<=15.0)
    {
        VecPosition posGoal( PITCH_LENGTH/2.0,
```

```
                    (-1 + 2*(WM->getCurrentCycle()%2)) * 0.4 * SS-
                                                    >getGoalWidth() );
        soc = kickTo( posGoal, SS->getBallSpeedMax() );
    }
    else if(disToGoalieOpp>15.0&&disToGoalieOpp<=25.0)
    {
        soc=dribble(angleToChose,DRIBBLE_FAST);
    }
    else
    {   VecPosition   posNr11=WM-
                        >getGlobalPosition(OBJECT_TEAMMATE_11);
        soc=directPass(posNr11,PASS_FAST);
    }
}//8 号攻击型队员的策略实现
else
if(WM->getPlayerNumber()==6)
{
    VecPosition   posNr8=WM-
                        >getGlobalPosition(OBJECT_TEAMMATE_8);
    soc=directPass(posNr8,PASS_FAST);
}//6 号配合型队员的策略实现
else
if(WM->getPlayerNumber()==4)
{
    VecPosition   posNr7=WM-
                        >getGlobalPosition(OBJECT_TEAMMATE_7);
    soc=directPass(posNr7,PASS_FAST);
}//4 号配合型队员的策略实现
```

```
else
if(WM->getPlayerNumber()==5)
{
    VecPosition    posNr4=WM-
                        >getGlobalPosition(OBJECT_TEAMMATE_4);
    soc=directPass(posNr4,PASS_FAST);
}//5 号配合型队员的策略实现
else
if(WM->getPlayerNumber()==3)
{
    VecPosition    posNr4=WM-
                        >getGlobalPosition(OBJECT_TEAMMATE_4);
    soc=directPass(posNr4,PASS_FAST);
}//3 号配合型队员的策略实现
else
if(WM->getPlayerNumber()==2)
{
    VecPosition    posNr4=WM-
                        >getGlobalPosition(OBJECT_TEAMMATE_4);
    soc=directPass(posNr4,PASS_FAST);
}//2 号配合型队员的策略实现
else
if(WM->getPlayerNumber()==10)
{
    disToGoalieOpp=WM-
                        >getRelativeDistance(OBJECT_OPPONENT_1);
    if(disToGoalieOpp<=15.0)
    {
```

```
            VecPosition posGoal( PITCH_LENGTH/2.0,
                    (-1 + 2*(WM->getCurrentCycle()%2)) * 0.4 * SS-
                                                >getGoalWidth() );
            soc = kickTo( posGoal, SS->getBallSpeedMax() );
        }
        else if(disToGoalieOpp>15.0&&disToGoalieOpp<=25.0)
        {
            VecPosition    posNr9=WM-
                          >getGlobalPosition(OBJECT_TEAMMATE_9);
            soc=directPass(posNr9,PASS_FAST);
        }
        else
        {
            soc=dribble(angleToChose,DRIBBLE_FAST);
        }
}//10 号攻击型队员的策略实现
else
if(WM->getPlayerNumber()==11)
{
        disToGoalieOpp=WM-
                          >getRelativeDistance(OBJECT_OPPONENT_1);
        if(disToGoalieOpp<=15.0)
        {
            VecPosition posGoal( PITCH_LENGTH/2.0,
                    (-1 + 2*(WM->getCurrentCycle()%2)) * 0.4 * SS-
                                                >getGoalWidth() );
            soc = kickTo( posGoal, SS->getBallSpeedMax() );
        }
```

```
            else if(disToGoalieOpp>15.0&&disToGoalieOpp<=25.0)
            {
                VecPosition    posNr8=WM-
                                >getGlobalPosition(OBJECT_TEAMMATE_8);
                soc=directPass(posNr8,PASS_FAST);
            }
            else
            {
                soc=dribble(angleToChose,DRIBBLE_FAST);
            }
        }
        else
        {
            VecPosition posGoal( PITCH_LENGTH/2.0,
                                (-1 + 2*(WM->getCurrentCycle()%2)) * 0.4 * SS-
>getGoalWidth() );
        soc = kickTo( posGoal, SS->getBallSpeedMax() ); // kick
maximal*///zjfroot
        }//11 号攻击型队员的策略实现
      ACT->putCommandInQueue( soc );
      ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
      Log.log( 100, "kick ball" );
    }
          else if( WM->getFastestInSetTo( OBJECT_SET_TEAMMATES,
OBJECT_BALL, &iTmp )
                == WM->getAgentObjectType()  && !WM->isDeadBallThem() )
    {                                   //如果自己与球的相对速度最大
```

```
            Log.log( 100, "I am fastest to ball; can get there in %d cycles", iTmp );
            soc = intercept( false );                    // 断球
        }
    return soc;
}
```

附录 6：实现守门员策略的 toxic_goalie()方法的源代码

```
SoccerCommand Player::toxic_goalie(  )
{
   int i;
   SoccerCommand soc;
   VecPosition   posAgent = WM->getAgentGlobalPosition();//获得球员位置
   AngDeg        angBody  = WM->getAgentGlobalBodyAngle();//获得球员角度

   // 定义矩形
   static const VecPosition posLeftTop( -PITCH_LENGTH/2.0 +
                                        0.7*PENALTY_AREA_LENGTH,
-PENALTY_AREA_WIDTH/4.0 );
   static const VecPosition posRightTop( -PITCH_LENGTH/2.0 +
                                        0.7*PENALTY_AREA_LENGTH,
+PENALTY_AREA_WIDTH/4.0 );

   // 定义矩形边界
                    static      Line      lineFront      =
Line::makeLineFromTwoPoints(posLeftTop,posRightTop);
   static Line  lineLeft  = Line::makeLineFromTwoPoints(
                            VecPosition( -50.0, posLeftTop.getY()), posLeftTop
);
   static Line  lineRight = Line::makeLineFromTwoPoints(
                                                  VecPosition(   -50.0,
posRightTop.getY()),posRightTop );
```

```
if( WM->isBeforeKickOff( ) )//守门员在比赛开始前的策略
{
   if( formations->getFormation() != FT_INITIAL || // not in kickoff formation
         posAgent.getDistanceTo( WM->getStrategicPosition() ) > 2.0 )
   {
      formations->setFormation( FT_INITIAL );      // go to kick_off formation
                              ACT->putCommandInQueue(     soc=teleportToPos(WM-
>getStrategicPosition()) );
   }
   else                                 // else turn to center
   {
       ACT->putCommandInQueue( soc = turnBodyToPoint( VecPosition( 0, 0 ),
0 ));
      ACT->putCommandInQueue( alignNeckWithBody( ) );
   }
   return soc;
}

if( WM->getConfidence( OBJECT_BALL ) < PS->getBallConfThr() )
{                                    // 如果看不到球
   ACT->putCommandInQueue( searchBall() );      // 寻找球
   ACT->putCommandInQueue( alignNeckWithBody( ) );
}
```

```
  else if( WM->getPlayMode() == PM_PLAY_ON || WM->isFreeKickThem() ||
                WM->isCornerKickThem() ) //扑球策略
  {
    if( WM->isBallCatchable() )
    {
      ACT->putCommandInQueue( soc = catchBall() );
      ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
    }
     else if( WM->isBallKickable() )
    {
       soc = kickTo( VecPosition(0,posAgent.getY()*2.0), 2.0 );
       ACT->putCommandInQueue( soc );
         ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc
) );
    }
    else if( WM->isInOwnPenaltyArea( getInterceptionPointBall( &i, true ) ) &&
                          WM->getFastestInSetTo( OBJECT_SET_PLAYERS,
OBJECT_BALL, &i ) ==
                                                              WM-
>getAgentObjectType() )
    {
      ACT->putCommandInQueue( soc = intercept( true ) );
      ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
    }
    else
```

```
    {
      // 在球和球门中心之间作一条直线
      VecPosition posMyGoal = ( WM->getSide() == SIDE_LEFT )
                            ? SoccerTypes::getGlobalPositionFlag(OBJECT_GOAL_L,
SIDE_LEFT )
                            : SoccerTypes::getGlobalPositionFlag(OBJECT_GOAL_R,
SIDE_RIGHT);
                        Line lineBall = Line::makeLineFromTwoPoints( WM-
>getBallPos(),posMyGoal);

      // 移动到该线上
      if( posIntersect.getDistanceTo( WM->getAgentGlobalPosition() ) > 0.5 )
      {
        soc = moveToPos( posIntersect, PS->getPlayerWhenToTurnAngle() );
        ACT->putCommandInQueue( soc );
        ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc )
);
      }
      else//朝向球
      {
        ACT->putCommandInQueue( soc = turnBodyToObject( OBJECT_BALL
) );
        ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc )
);
      }
    }
```

```
    }
    else if( WM->isFreeKickUs() == true || WM->isGoalKickUs() == true )//球门球策略
    {
        if( WM->isBallKickable() )
        {
            if( WM->getTimeSinceLastCatch() == 25 && WM->isFreeKickUs() )
            {
                //移动到前方对方球员较少的位置。
                if( WM->getNrInSetInCircle( OBJECT_SET_OPPONENTS,
                                                    Circle(posRightTop, 15.0 )) <
                        WM->getNrInSetInCircle( OBJECT_SET_OPPONENTS,
                                                    Circle(posLeftTop,  15.0 )) )
                                                        soc.makeCommand( CMD_MOVE,posRightTop.getX(),posRightTop.getY(),0.0);
                else
                        soc.makeCommand(CMD_MOVE,posLeftTop.getX(),
                                                            posLeftTop.getY(), 0.0);
                ACT->putCommandInQueue( soc );
            }

            }
        }
    return soc;
}
```

外文资料翻译

Karlsruhe Brainstormers – 机器人足球中的再励学习方法

摘要：我们长远的目标是建立起一支完全基于再励学习方法决策的机器人足球队，这篇论文描述了 Karlsruhe Brainstormers 仿真足球队所追求的一般性方法。基本决策方法的主要部分是由再励学习方法解决。在策略层，展示了 2 对 2 进攻情景下的经验结果。

1 引言

Karlsruhe Brainstormers 在 robocup 仿真组领域的努力的主要目的是去研发并应用再励学习技术于复杂领域中。我们长远的目标是建立一个学习系统，只需要使用“赢得比赛”作为输入，智能体便可以学习生成合适的场上动作表现。在机器人足球领域，22 个队员和球被允许拥有达 $(108\times50)^{23}$ 种的位置,并且如果考虑到物体速度和球员体力这些状态话,完整的状态空间大小是上述数值的指数级的倍数。在服务器的每一个循环中,即使一个无球的智能体也有 300 个基本命令(包括参数化的转身和加速)可选。这样一来使球队在每一个循环有 300^{11} 动作可选。这种复杂性对当今的再励学习方法来说是一个非常大的挑战。在 Brainstormers 的项目中我们对为在这种大规模尺度上去实际的

解决学习问题所采用的多种方法进行了研究。

2 机器人足球中再励问题的体现

我们在机器人足球比赛中面临的问题可以阐述为以下问题：在 Markov 决定问题（MDP 问题） $M=\{S,A,p(.|.),c(.)\}$ 中寻找一个最优方法 π^*。其中状态空间 S 中的一个状态 s 由 22 个队员和球的位置和速度构成；A 中的一个动作 a 是一个基本命令踢、转身、加速（对每一个队员）的 22 维向量。世界模型 $p(s_{t+1}|s_t,a_t)$ 提供了当状态 s_t 被施以动作向量 a_t ，转换状态 s_{t+1} 的转换概率。最后，每一个转移过程产生一定消耗，这个过程指动作 a 被应用于状态 s 上的过程。由于我们通常情况下对只控制一个球队感兴趣，所以我们将假设在动作向量中，只能选取 11 个与我们球队相关的条目。在论文的余下部分，我们将认为剩下的 11 个对方球队的队员是被任意选取，并且是未知的，不过我们同时认为对方球队采用的策略是不变的。MDP 问题的任务是寻找一个最有的方法 π^*:S→A。,使在整个决定过程中的所选路径：$J^*(s)=min_\pi \ E\{\sum_{(t=0)} c(s_t,\pi(s_t))|s_0=s\}$ 所积累的预期消耗最小。球队的中心策略 π^*将必须能够分配给每一个状态一个 11 维的命令向量。

再励学习在原理上被设计成为，在状态空间极大的情况下以至于传统方法无法解决时的，来寻找 MDP 问题的最优解或近似最优解。其基本方法是通过反复地从 $J^*=J_{k+1}(s_t)\leftarrow\{c(s_t,a_t)+J_k(s_{t+1})\}$ 中确定 J^* 来近似接近最优值函数。然而机器人足球队领域过于复杂，以至于当前已知的再励学习算法无法在可以接受的时间内得到最优解。因此下面我们试着给出一个对我们所面临的问题的分析以及已解决和仍在公开寻求解决方案的讨论。总的来说，需要解决

以下两个方面的问题：其一，复杂度需要降低；其二，解决在分布式系统中的再励学习问题。

2.1 复杂度的降低

从原理上，复杂度有三个来源：状态数量，动作数量，在一个求解路线结束之前需要作的决定的数量。再励学习方法背后的一种思想是通过对外界互动产生的经验的学习来处理复杂的状态空间。因此通过只考虑于状态空间相关的部分，可以使复杂度得到削减。复杂性的进一步削减则可以通过对特性学习而不是学习直接的状态空间得到。不过这样以来将把 MDP 问题转换成为部分可观测的 MDP 问题（POMDP 问题），而这个问题目前还属于一个正在研究的课题。动作的数量的减少可以通过任意限制智能体可选择的命令的数量来实现。在某种极端情况下，我们可能只允许仅仅一个动作可选，不过这将使决策变得无用，但我们可以从这个极端中明显的看到可选动作的减少，讲意味着解的质量的下降。因此避免它的发生的技巧在于尽可能多的使用一个好的策略下的动作，而不是尽量使用越多的动作越好。在基本命令层上决定哪一个基本动作适合特定的情况是非常困难的。然而如果我们考虑动作序列而不是单独的动作，问题将得到简化:球员需要一个序列的动作来实现断球、带球等动作，我们称这个序列为一个步骤（mov），而每个步骤有一个定义好的子目标。步骤的概念的重要性体现在以下两个方面：它有效的减少了智能体可选择的动作数量，并且通过引入基本命令序列减少了智能体需要作的决定数量。而步骤本身，就像我们下面可以看到，可以通过再励学习方法进行学习得到。

2.2 分布式系统中的再励学习

通过使用步骤的概念，智能体策略的选择范围从一系列的命令变为一系列的步骤，我们称这种方法为灵活决策。在机器人足球领域，这种决策需要由每一个智能体独立来完成。由此引出了分布式场景中的学习问题。由于所有的智能体拥有一个共同的目标，因此这是一个合作的多智能体系统。在MDP 问题框架中，合作的方面可以通过分配给每一个智能体相同的转换消耗c()来进行建模。有限的通信能力引出了进一步的难题：智能体不知道它的队友所进行的动作，只能仅仅通过观测决策的结果状态空间来推测。这种方案通常被称为“独立学习者（IL）”方案。在假设的一个固定的环境中，我们最近为以 IL 为智能体的球队提出了一种分布式学习算法。该算法向随机环境的扩展也正在进行开发之中。这将为独立行动的智能体队员的学习提供理论基础。一些基于启发式学习来代替单智能体 Q－learning 的方法的实验已经进行，并得出了实验结果。

3 步骤

一个步骤是一系列基本动作组成的一个序列，经过若干时间周期之后是从目前状态 $s(0)$转移至新的状态 $s(t)$。带来的结果是一组终止状态 S^f，并且有可能是积极状态 S^+或消极状态 S^-。在达到终止状态 $s(t) \in S^+$时，或者时间超过最大时间限制 t_{max}时，一个步骤结束。

举例说明：断球的步骤在两种情况下结束：要么球在队员在可踢的范围内，要么队员遇到了一个没有接触到球的可能性的状态 S^-。

3.1 步骤的再励学习。

既然每一个步骤都有一个清楚定义的目标，那么目前的任务就是去找一个基本命令的队列以完成目标工作。这项工作可以通过传统的方法进行编程实现，也可以像我们工作中使用再励方法进行。再励学习的通用方法如下：智能体仅仅被告知它的行动的最终目标，以及可以使用的所有可能动作，在这里的话就是转身、踢球、加速等动作。每一个时间步中，会有一个命令根据情况被选择。学习在这里的意思是通过逐渐改进决策策略以使学习目标的完成能够越来越好。这里我们使用实时动态的编程方法，通过重复尝试增加近似最优值函数来解决学习问题。由于状态空间是连续的，于是一个反馈神经网络被用于近似值函数。若需要对神经网络中再励学习方法的使用作详细的了解，请参阅[3]。

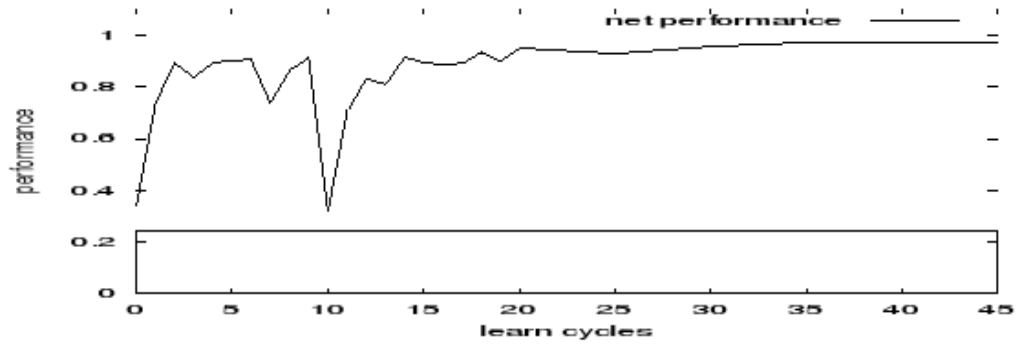


图 1

图 1. 学习使用最大速度 2.5 m/s 踢球的动作的效果的一个例子。每一个学习周期由 2000 个初始状态组成，每个初始状态包括不同的球的位置，速度

和相应的运动轨迹。在大约 15000 个单独学习周期更新后，网络稳定下来，达到 95%的性能。

例：学习踢球 寻找一个好的踢球路径是一个非常冗长的工作。在再励学习框架中，这项工作由智能体自身所完成。智能体踢球的命令提供有 500 个参数化的实例（方向被离散化为 100 步，力量则被离散化为 5 步）。再加上 36 个转身命令的实例，在一个周期中一共将有 536 个动作供每个智能体选择。学习的目标就是寻找一个踢球和转身的序列，以使最终求被射出的时候能够以满足所要求的速度和方向。这个目标定义了积极状态 S^+ 。如果这样的一个状态达到了零消耗，则该序列退出。然后声成一个消极状态 S^- ,在队员在该队列中丢失球的情况发生时，产生最大消耗 1。由于一个合理的优化目标是用最可能少的命令产生一个成功的序列，所以每个中间的转换产生固定消耗 $c(s,u)=0.002$ [3]。在两个小时的学习后，得到的结果策略非常复杂，比手工编写策略有效的多。智能体学会了把球拉回自己身边，自己转身(如果需要的话)，加速多次以产生高速度(如图 1)。通过学习可成功地加速到 2.5 m/s。总体上一共有三个不同的神经网络用于训练，一个目标速度上限为 1.0m/s，第二个目标速度上限 2.0m/s，第三个目标速度上线 2.5m/s。(每一个神经网络均使用 4 个输入，20 个隐藏的和 1 个非隐藏的输出神经元)。

4 策略层

在策略层上，每个智能体必须在以下活动中作出决定：断球，在 8 个方向中的 1 个前进，在当前位置等候，传球给队友(有 10 个选择)，3 种射门方式，在 8 个方向中的 1 个盘球前进。无球情况下总共有 10 个动作情景，有球

情况下有 22 个动作情景。没有智能体可以独立赢得比赛，因此智能体之间必需学会合作并协调他们的表现。有效的策略是指每个智能体表现出可靠并对共同目标进行行动。

4.1 球队策略的再励学习

我们初始的实验在 2 名进攻队员对 1 至 2 名防守队员的情况下实施的。防守队员有一个固定的策略：跑向球，并且如果得到球，则将球踢向进攻者的球门。进攻者的任务是尽可能快的得分。如果进球成功，进攻队员将被奖励 0，如果失去球将被惩罚 1。每一个时间步进攻者被赋予固定的 0.002 的值直到进球。这意味着需要 500 个时间步才会使当前策略和丢球一样坏。每一步均需要做一个新的决定，并且智能体有球时(可能给 1 个人)有 13 个动作可选，无球时有 10 个动作可选。与[4]中的进展相比，我们使用完整的连续状态信息作为输入。既然状态信息是连续的，我们使用神经网络来表示值函数。在最终的实现中，每一个动作均使用一个独立的网络来表示。如果这个动作运用于不同的状态产生不同的相应的开销，这就要求允许对每一个动作使用独立的特性，支持通用的能力。以下表格展示与贪心算法想对比下学习策略的改进，学习的内容是得到球，并试图射门得分。

贪心策略在有 1 个防守队员时 35%的情况下可以得分，在有 2 个防守队员时，只有 10%的情况下能够得分。再励学习策略下，2 个智能体成功地学习了在大多数情况下如何配合以成功得分。在有 1 个防守队员时，85%的情况下可以得分，在 2 个防守队员时，55%的情况下可以得分。这是一个非常高的胜率。因为在正式的比赛中，不进球并不意味这我方失去对球的控制权，

我们甚至观察到了一些复杂的配合，比如二过一的配合。

5 总结

机器人足球领域可以被模型化为一个复杂的 MDP 问题，主要的特点是高维数，连续状态空间，巨大的基本命令数量以及对再励学习和部分可观测状态信息问题来说需要的巨大的策略数量。

目前 Karlsruhe Brainstormers 解决方法是使用步骤来尝试解决复杂度问题，并且这些步骤通过使用再励学习方法可以学习得到，并可以定义一个智能体的技能。使用前馈神经网络来近似连续状态空间的函数。在处理多智能体方面，我们既追求理论的分布式的再励学习算法的研究，又考虑经验的，启发式的单智能体 Q-learning 的学习方法。然而仍然有许多研究问题未解决，比如当处理部分可观测状态信息时，理论上的定义和有效的分布式学习算法，值函数的理想表示方法，寻找最优近似特性和使用它们的理由，自动寻找近似的动作等等问题。不过，再励学习方法已经被证明在我们的智能体对抗中十分有效。

5.1 在球队对抗中的再励学习方法

在目前的对抗中，Brainstormers 球队的智能体所有的基本步骤(move)都是由再励学习方法来学习得到。比如：1.可以在一个需要的方向上将球从 0 加速到 2.5m/s 的踢球(kick)动作。2.使用随机域的有效断球的断球动作(intercept-ball)。3.带球跑动中保持对求的控制的带球动作(dribble)。4.到达指定地点以避免与其它球员相撞的跑位动作(positioning)。5.停止高速过来的球的停球动作(stop-ball)。6.防止对方球员断球的护球动作(hold-ball)。基本上所

有的基本命令均使用神经网络的决策方法进行制定。

在策略层，我们无法将 2v2 的结果直接运用于实际的比赛对抗中。目前使用的是一种被称为中级再励学习方法的策略：每一个步骤被判断其可用性和成功的可能性。可用性好坏的判定由被称为 PPQ 判定的简单优先排序方法得到。

5.2 致谢

我们非常感谢 CMU 球队提供部分源代码，我们使用了其世界模型部分。

外文资料翻译原文