THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY 2052 · AUSTRALIA

The University of New South Wales

School of Computer Science & Engineering

Road to RoboCup 2006

rUNSWift 2006 Behaviors & Vision Optimizations

by

Enyang HUANG

Submitted as a requirement for the degree

Bachelor of Software Engineering with Honors

Submitted: 1 September 2006

Supervisor: Dr William Uther

Assessor: Professor Claude Sammut

Abstract

This thesis describes work done in visions and high-level behaviors in 2006 at rUNSWift[1]. For both parts, we have introduced new approaches and re-implemented bunch of the modules. The first part of this thesis, Vision, covers the 2006 implementation of chromatical correction of raw images with machine learning approach. Moreover, as a result of the new rule on the field, we also introduced the concept of field edge, which is used by higher level agent vision processes. The second part of this thesis is a review of the 2006 rUNSWift behavior system. Lots of the modules and concepts here are, re-defined carefully and re-implemented, with concentration from simple low-level skills such as ball tracking, ball finding, ball grabbing, ball dribbling, ball kicking, to high-level complex behaviors such as role assignments, positioning and team cooperation. In parallel, a strategy simulator was implemented which is capable of performing plug-in-and-play simulation of the behavior modules.

The team comes 1st in Australian Open 2006[2], and 2nd in Robocup 2006 Bremen[3], using 2006 rUNSWift code base.

[1]rUNSWift is a combination entry of the University of New South Wales and Nation ICT Australia in Robocup 4 legged-league. National ICT Australia is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

[2]See http://www.cse.unsw.edu.au/robocup

[3]See http://www.robocup2006.org

Contents

List of Figures

Chapter 1

Introduction

In the first week of summer holiday 2006 (December, 2005), the new 2006 rUNSWifters started their journey. 2006 rUNSWift directly inherited the spirit of rUNSWift 2005. In particular, the basic approaches of vision, locomotion, behavior and system architecture are similar in concept and structure comparing with year 2005. Based on the weakness of team 2005, the team has proposed many optimizations. Large-scale work was conducted in modules related to localization, low-level visions, and behavior. Alone with these lines of work, the team has also developed some offline tools such as offline vision, offline behaviors simulator.

In the following chapters I will introduce and discuss the work that I was in charge. These consist of topics mainly in vision and behavior modules.

1.1 Background

Robocup is an International research project to promote AI, robotics and related field. In the 4 legged league of Robocup consists of 4 Sony Aibo robots playing on a 540*360 field. The robots must operate autonomously and no hardware modification is allowed.

rUNSWift has had good record in 4 legged league of Robocup since its first participation in year 1999.

1.2 rUNSWift 2006 Architecture

The rUNSWift 2006 architecture consists mainly of censoring/processing, behavior, and actuator. The system is running 30 times a second. Thus each frame has 1/30 second of time to sense its environment, update states, making decision and react. The system was implemented efficiently thus

most of the time a full frame process is completed in time.

Sensors used at rUNSWift consist of OpticalRadiation Sensor (Head Camera, HeadChest Infrared) and Mechanical Sensor (Paw Touch Sensor, Head Parameters Sensor). Environment censoring is done primarily by Head Camera. Head camera produces raw image every frame. The following processing also consists of two folds. We first apply image process to extract information out from the raw image. Secondly we update agent's state. For example, we do localization update (See Oleg's Thesis), Obstacle map update (see Josh's Thesis), as well as storing the objects and their attributes handily.

When censoring is completed, we move on to behaviors processing. The structure of this stage naturally consists 3 steps. The first step looks at the information provided by current frame censoring, as well as those information from previous frame left in "memory", makes correct decision on what roughly action the agent should perform. For example, at this stage, the agent might think if it needs to walk forward or backward, or if it needs to kick or release the ball. This state is therefore called decision. It may decide a action to perform only in a particular single frame, or may also decide a sequential actions that need to performed over the next few frame, depending on the exact circumstance. Once the next-step decision is decided, the agent looks at the skills it knows how to perform, selects the most appropriate one, and executes that skill according to how it is specified. The final step is the communication. Here the agent reports what has done to its team. Communication and sharing information happens not necessarily at the end of run of the behaviors. For example, in order to make the correct next step decision, an agent might "ask" its team mate a few questions and "negotiate" before making the final decision.

The physical effect of execution of a particular decision is usually the movements of agent's actuators. Either single one such as raising the tail, or movement of multiple ones such as moving four legs simultaneously to simulate walking (See Ryan's Thesis).

The above software structure can be graphed as 1.1:

As can be seen from the graph, there are 5 important sub-systems, Vision, Localization, Behavior and Actuator Control.

[Vision System] - This system processes YUV image from head camera at frequency of 30 hz, using sub-sampling methods. Interesting features are recognized and object recognition is performed over these feature. These objects include goals, beacons, ball, field lines, and field edge. A detailed

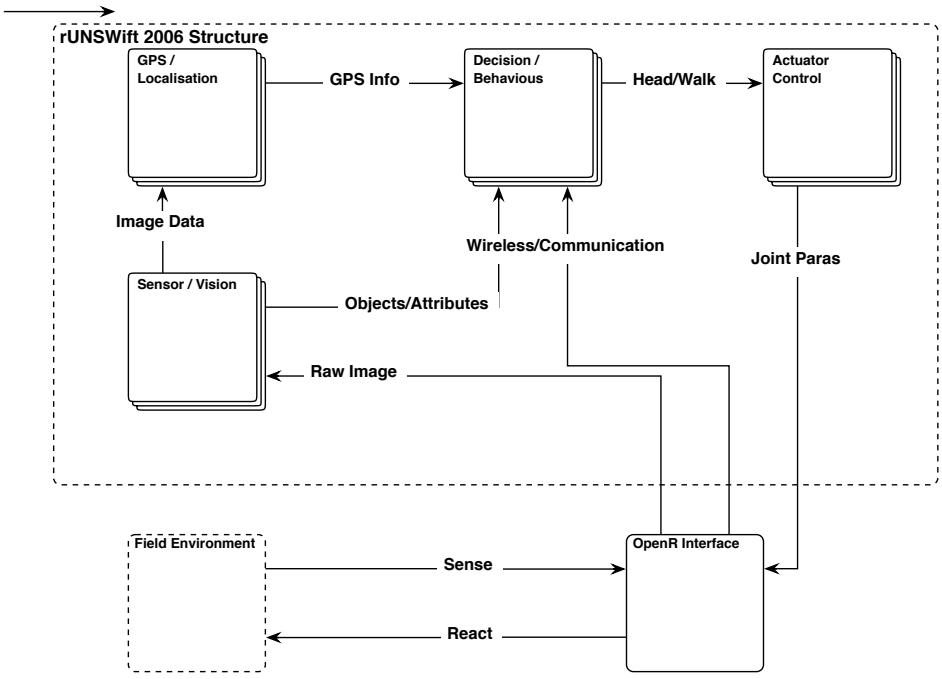

Figure 1.1: The rUNSWift architecture react with outside environment through OpenR interface. Agent at each frame sense and then react. Precisely, it first senses the world, store the results after processing image, update localization, make decision and finally set the joint parameters at the end of that frame.

desciption of this year's vision system please refer to [1].

[Localization] - This module tracks the position of the important objects on the filed. They include the robots, ball and obstacles. It uses a extended multi-modal Kalman Filter for self localization and updates its position with new observation of landmarks. Field lines are used as sanity checks over these procedures. A detailed description of this year's localisation is at [2].

[Behaviors] - This system uses information provided by vision system and localization system, together with wireless communication with teammates, selects the best strategy to perform in real time.

[Actuator Control] - This system handles locomotion of the robots. It receives commands from behavior system and move the corresponding raw joints appropriately. For detailed description on Actuator Control modules please see [3].

1.3 Report Overview

This thesis will explain work details in part of 2006 Vision system. In particular, low-level vision and field edge recognition in high level vision. This thesis will also give a comprehensive overview of the entire behavior system. In particular, behavior 2005-2006 diff will be discussed in more detail. Chapter 2 will describe the 2006 camera chromatical distortion correction approaches. In chapter 3 we will discuss this year's field edge recognition using extended Ransac algorithm. From chapter 4 on, we will present the description of the entire behaviour layer in our architecture. Chapter 4 introduces the behaviour system structure and high-level behaviour modules including role, role assignment and role positions. Chapter 5 describes the strategy decision tree that links between high-level behaviour modules and low-level behaviour modules. Chapter 6 introduces our low-level behaviour system, with emphasize on the four important skills - ball tracking, ball finding, ball grabbing, and ball dribbling. Chapter 7 is a description of this year's development on behaviour simulator - its design and application. The last part of this thesis summarizes the work and present the outlooks for future works.

Chapter 2

Ring Correction

2.1 Overview

A significant problem introduced by the Sony AIBO ERS-7 robots is the chromatic distortion near the edge of the image. Chromatic distortion happens when different colors of light do not line up across the entire image plain, and is usually dependent on the lens aperture size. [5][6] Some other source also claims that the distortion can be difficultly explained merely in terms of optics and suspect it could be partially due to digital effects[7]. The distortion looks roughly in cyclic shape and as it distanced away from the center, the distortion is more apparent. In the domain of Robocup 4 legged league, this problem is common to many teams. At rUNSWift, colors classification plays a fundamental role in the vision system. High-level vision relies directly on the accuracy of low-level colors classification. Thus, this problem needs to be resolved.

Because this effect looks closely cyclic, so it is termed "Ring Effect" at rUNSWift. The correction method developed here is therefore called "Ring Correction".

2.2 Past Work

Since the introduction of ERS-7 model, many Robocup teams attempted to come up with effective solution to this ring effect. Early work at rUNSWift was conducted by Jing Xu in 2004. Jing Xu concluded the following characteristics of this ring effect [8]:
The chromatical distortions experienced by AIBo ERS-7 are:

(a) Symmetric about its center, which is roughly, although not exactly, the center of the image.

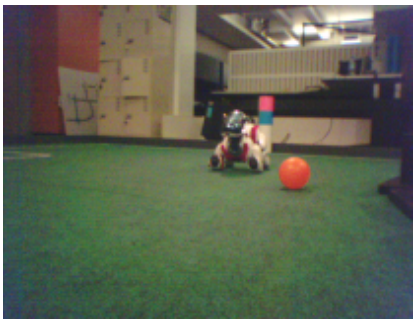

Figure 2.1: A standard picture taken from ERS-7's camera without correction. We can notice the pink fringe near the edge of the picture.

Figure 2.2: A picture of an uniform white wall taken from ERS-7's camera without correction. One can notice how severe the distortion is as the distance is awayed from the center.

(b) Parabolic in terms of its relationship between chromaticity and pixel position.

(c) Different for each distinct color.

(d) Roughly consistent for different colors with the same camera settings.

(e) Independent of lighting conditions.

It seems that if the distortion can be modeled, then it can be corrected. Due to the uncertainty of the cause of such distortion, one may not be able to express such distortion function in a closed formula. If the distortion model if F, then to correct a image with distortion F(image), we can apply the inverse of F, $F^{'}$, hence to correct, image = $F^{'}$ (F (image))

Jing Xu eventually ended up with a model similar to electric fields. Given a pixel's value, and its position on the image, the pixel's value before distortion can be calculated from solving a system of quadratic equations[8]. In the following section, we will present this year's correction algorithm, which is much faster to implementation than the 2004 model.

2.3 Presentation of the work

We now need to develop an algorithm to solve the ring effect. The input of the algorithm will be an arbitrary pixel on a YUV plain, with its values and

positions. The output of the algorithm will be the YUV value of the same pixel after ring correction, so the distortion effect is minimized. Apart from these, the implementation of the algorithm must be efficient, recall that this algorithm runs at the very bottom of our architecture, hence its speed is extremely vital. The efficiency will also be stressed in the following content.

We use YUV color model to describe a pixel on one image frame. Where Y is the standard luminance component, also called brightness. U and V are the chrominance components, also called color components. Early work done by Jing Xu suggested that when distortion of a pixel happens, that distortion exists on all Y, U and V components. However, it is unclear that there is any correlation among the individual distortion components. The plus side of assuming Y, U and V are independent of each other is that its implementation is considered to be computationally efficient. [8] Early concept built based on this YUV independency worked well in practice. Hence this year's approach our model also assumes Y, U and V is independent at distortion. This naturally leads to the thoughts of building three models, each for Y, U and V.

2.3.1 Theory

We start by observing these effects first. We took pictures of uniform colors from the robots. By uniform we mean same or close to same pixel value on the real object, such as a pure white board. See figure 2.4,2.4,2.4,2.4

The chromatic distortion of component Y (We take Y as an example, U and V are similar.), given a reference spectrum, is dependent on its magnitude. Moreover, it also depends where it is on the image. We therefore use its distance from the physical center of the image as a measurement.

We then decided to use a polynomial approximation function F to model Y component distortion. The forward distortion model for Y component is a function F mapping from the correct value to its corresponding distorted

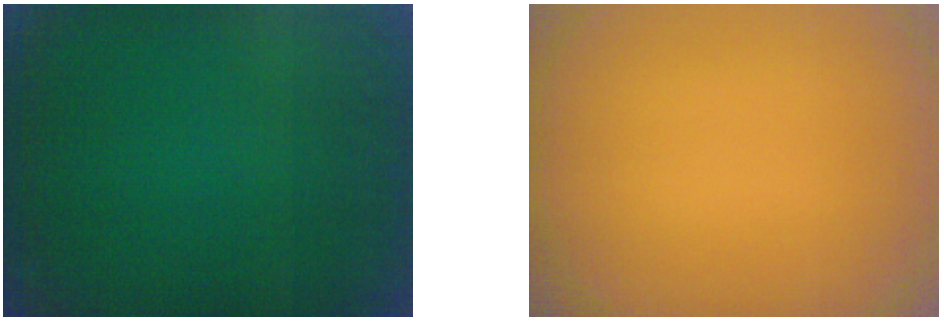

Figure 2.3: A pictures by ERS-7 of pieces of papers of pure colours. TopLeft: Pink uniform colour paper raw image. TopRight: White uniform colour paper. BottomLeft: A green Uniform colour paper. BottomRight: Yellow uniform pirce of paper

value:

$$F : Y_c \rightarrow Y_d \tag{2.1}$$

Given the distance from this pixel to the image center d_Y, the full polynomial model we used is given by:

$$Y_d = (a_1 Y_c + b_1)(d_Y)^3 + (a_2 Y_c + b_2)(d_Y)^2 + (a_3 Y_c + b_3)(d_Y)^1 + Y_c \tag{2.2}$$

To apply correction, we use F^{-1}, that is:

$$F^{-1}(F(Y_c)) = Y_c$$

In particular with our polynomial distortion model, we have:

$$F^{-1} = \frac{Y_d - b_1(d_Y)^3 - b_2(d_Y)^2 - b_3(d_Y)^1}{a_1(a_Y)^3 + a_2(a_Y)^2 + a_3(a_Y)^1 + 1} \tag{2.3}$$

2.3.2 Function Minimization

Now to come up with these coefficients, we need apply minimization algorithm. That is, given pixel value (y,u,v) and its distance from the center d_Y, $F(value, d_Y)$ can be as close to the expected magnitude as practical.

In our experiment, we have selected a set of uniform color sheets of papers. The color we used in the experiment are: blue, green, darkblue, pink, white, yellow. Of course, one can selects more colors. At the next step we take pictures on each of these objects to form out a training set χ.

The coefficient matrix to be determined is:

$$Coef_F = \begin{pmatrix} a_1^Y & b_1^Y & a_2^Y & b_2^Y & a_3^Y & b_3^Y \\ a_1^U & b_1^U & a_2^U & b_2^U & a_3^U & b_3^U \\ a_1^V & b_1^V & a_2^V & b_2^V & a_3^V & b_3^V \end{pmatrix}$$

Let F_Y be the distortion model of Y component on a pixel, that is, the coefficients for F_Y is the first row in $Coef_F$. Similarly, F_U and F_V will use the second and third row respectively. The error function Y colour component per pixel is given by:

$$pixelERR_Y = (F(Y_{center}, d_Y) - Y)^2 \tag{2.4}$$

That is, the Y component error at this pixel is defined to by the square of the difference between the estimated Y value and the actual Y value appears in the picture. The Y value estimation could be calculated from the Y value of the center pixel in this image and the distance from the current pixel to the center pixel. In addition, the total error at this pixel is defined as the sum of all component errors:

$$pixelERR = pixelERR_Y + pixelERR_U + pixelERR_V \tag{2.5}$$

Finally, the total error of our function at one step of optimization with entire χ is then the sum of all pixelERR in all picture. Function minimization will selects $Coef_F$ such that this total error is minimal. In the lab, we used the standard Simplex Algorithm[9]. Other popular optimization algorithms were used by Robocup peers. For example, Thomas and Walter at German Team used Simulated Annealing[10] in 2005, and ended with good results of distortion correction.[7]

The final $Coef_F$ returned from minimization is:

$$\begin{pmatrix} -8.497e^{-8} & 1.120e^{-5} & -5.909e^{-6} & -1.064e^{-3} & -1.577e^{-3} & 4.898e^{-2} \\ -1.561e^{-7} & 1.081e^{-5} & -3.532e^{-6} & 1.331e^{-3} & -1.145e^{-3} & 1.363e^{-1} \\ -1.331e^{-7} & 3.059e^{-5} & -1.431e^{-5} & 9.920e^{-4} & -4.731e^{-4} & 1.033e^{-1} \end{pmatrix}$$

2.4 Implmentation

The whole implementation would follow two parts: learning model and correcting. Both were implemented successfully at rUNSWift.

The correcting procedure is rather simple, but need to be implemented efficiently. Look up table seems to be the natural choice in the context.

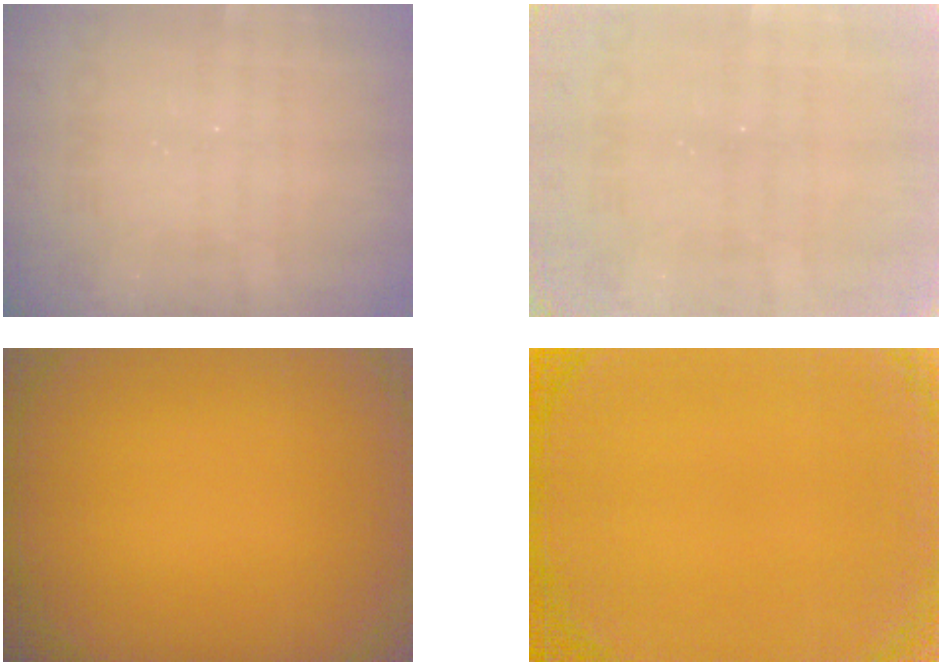

Figure 2.4: The images before ring correction and after ring correction. First Row: A picture by ERS-7 of a piece of pure white paper,Post distortion correction of a pure white image. We can see the ring effect has been minimized by significant level. Second Row: A picture by ERS-7 of a piece of pure yellow paper,Post distortion correction of a pure yellow image. The minimization quality was very high on this colour, comparing with the image before correction.

Because the input of the correcting function requires only several discrete values, the current Y, U and V value ranging from 0 to 255, and its distance from the center. On a 208 $\times$ 160 image frame. Here we want to convince the reader that our early claim of assumption of YUV independence helps increase the efficiency here.

Assume Y, U and V are not independent, and also assume our prelearned model matches YUV and its position from the center to Y'U'V'. Then it follows that there are (Size $\times$ Width $\times$ Y $\times$ U $\times$ V) cells needed to be stored. This gives us approximately: $208 \times 160 \times 256 \times 256 \times 256 = 5 \times 10^{11}$

Assume Y, U and V are independent of each other, thus split the original table into 3 smaller tables, this gives us (Size $\times$ Width $\times$ (Y + U + V)) cells, which is approximately 2.5×10^7

The last model assumes not only the independency among Y, U and V,

Figure 2.5: A standard image taken by ERS-7 on Robocup field with usual environment, the ring effect was obvious. The post distortion correction of the original image with our lab background. This is a image of non-uniform colour and is corrected well by our method.

but also take into account of the symmetry of the position. For example, the top left pixel would be treated as same as bottom right pixel. The concept is to use the distance between its actual position on the image and the image center. This further reduces our look up table to Distance $\times$ 256 $\times$ 3 = 132 $\times$ 256 $\times$ 3 = 1.0 $\times$ 10^5

Sony AIBO ERS-7 uses memory stick to store programs and instruction. The memory stick is often of size 16MB. Assume we use a char to represent Size, Width, Distance, Y, U and V. (Notice on typical platform a char is 1 byte) The first look up table would result in 1560GB. The second mechanism requires 25MB. The last look up table requires approximately 200KB.

In practice we used the last type of look up table, which consists of three look up table, each of capable in correcting Y, U or V component of a pixel. Each look up table is of type Correct[Int][Int], where the first entry is the actual value of the YUV value, from 0 to 255, the second is the distance value, from 0 to 132. The total look up table is of about 476KB in size, and capable of performing ring correction in virtually constant time.

2.5 Summary

The existence of the "ring effect" has presented a challenge to many teams. Various methods were used by different teams. The first approach is to ignore the problem. Seeing the chromatical distortion is only severe at the near edge of the image, those team only apply object recognition at the region they are confident about. A few teams also provide plausible solutions based on model and inverse model approaches, where people come up with some form of approximation of the distortion model. The correction model

is then the inverse function of the original model (rUNSWift04-06, NuBots, German Team).

Robot's vision must be accurate, robust, and efficient. [4] The framework we provided is accurate, robust and efficient. It only takes a few minutes to run the learning, and is capable of generating correction table regardless of lighting condition. The performance of this correction method is satisfactory from both experimental results in our laboratory, as well as from competition results with other peer teams.

Chapter 3

Field Edge Recognition

3.1 Overview

In year 2006, there will be no longer field wall in a competition. The existence of field wall can help detecting the field boundary, hence features above such boundary can be confidently ignored. We have developed a simple and efficient method that is able to recognize field edge in absence of the wall thus provides means of vision sanity checks for upper level processes. This sanity check is very important, in particular, there will be audience wearing colorful clothes close to the field in 2006.

Object recognition works by first constructing scanlines on the images. Along with the scanline, interesting features over pixels are marked. In addition, related features are then grouped to form object such as ball, goals and beacons. This year we introduced wall features, whereby we looking for boundary wherefrom absence of green color starts. Intuitively these are likely the boundary of the field from robot's perspectives. We then approximate a 2 dimensional line from these features. If the line fits well with regards to these features we accept it, otherwise the process fails.

Being able to accurately identify field wall is very advantageous. The first attempt of wall approximation at rUNSWift was by Nobuyuki Morioka in year 2005. [11] Another popular method used by our peer teams is the concept of "horizon". Horizon represents a line through the image with constant elevation equal to that of the camera. In practical, visual objects such as ball and goals rarely leaves the ground. Hence both "field wall" and "horizon" are good sanity checks to justify object recognized and have relatively high elevation. For example, an orange T-shirt on an audience might be falsely detected as ball. At Robocup 2006 Bremen, blue window directly behind the field might be falsely detected as a blue goal. Field wall and horizon are very effective when dealing with fake objects at wrong elevation

Figure 3.1: A typical image taken by our robots at World Championship 2006, Bremen. The image has complex backround patterns which can cause serious recognition problems. In this image it is likely our calculate of the goal position and size could be contaminated by the blue windows and lights on the background.

level. The drawback of horizon is that it is noisy, and can result in significant inaccuracies in the estimated result on the image.[4] [12] rUNSWift 2006 uses both horizon and field wall techniques in its sanity modules.

3.2 Present of Work

3.2.1 Detecting wall feature

The 2006 wall feature detection is similar in concept to the 2005 feature recognition developed by Alex North. A state machine tracks the number of green and non-green pixels encountered while walking through vertical scanlines. A wall feature is detected on a vertical scanline if a green to non-green transition is detected, and in particular the following are true:

(a) No wall feature was detected on the same scanline before, and
(b) Within last N pixels, the accumulated ratio of green to non-green is greater than a threshold

Then a wall feature is created at the current scanline position - half of N.

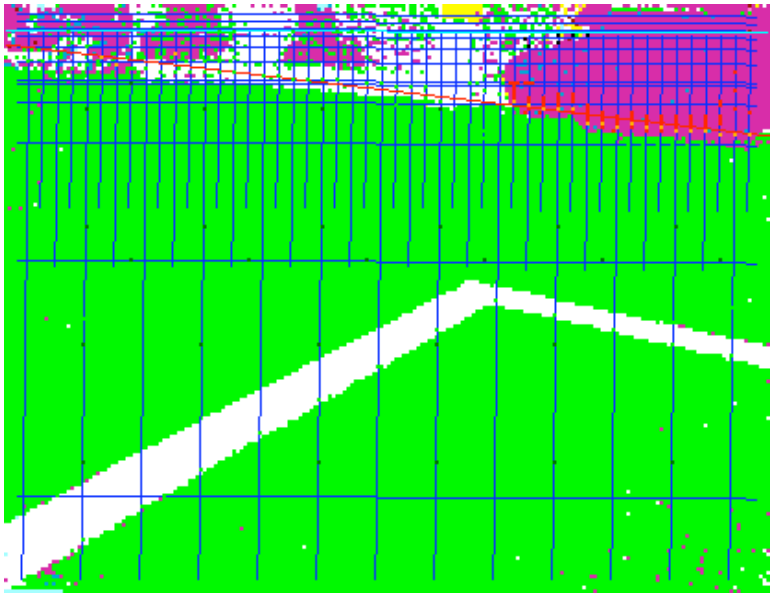

Figure 3.2: A field wall line marked on C plane of a vision frame. Notice the wall features marked on each vertical scan line with yellow dots. The line (red line) is accurately marked along the boundary of green to non-green transition boundary.

3.2.2 Random Sample Consensus

Having wall features detected, we now move on approximating of the wall line. The existance of the noise in detecting false wall features presented us with a difficult problem. Can we robustly and accurately estimate the wall line when outliers are present.

The team then decided to use a modified Ransac algorithm (Random Sample Consensus)[13] to overcome this problem. Ransac was first published by Fischler and Bolles in 1981, it is able to do robust estimation of the model parameters when dealing with high order of magnitude of noise. The disadvantage is that there is no upper boundary on time it takes to compute such fitting approximation. There is known faster algorithm similar to Ransac, which guarantees the same confidence of the solution as Ransac, but discard large number of erroneous model parameters in its preview procedures at each iteration, thereby increases its efficiency.[14] Here we only used a standard Ransac but with a time limitation on its runtime. The algorithm is presented with figure 3.1

Algorithm 3.1: The Random Sample Consensus[16]

Data: A set of observed data points
Input: n - the minimum number of data values required to fit the model
k - the maximum number of iterations allowed in the algorithm
t - a threshold value for determining when a data point fits a model
d - the number of close data values required to assert that a model fits well to data
Output:
Result: A model that can fit to data points well

```
1  iterations = 0
2  bestfit = nil
3  besterr = LargeValue
4  while iterations < k do
5  |  maybeinliers = n randomly selected values from data
6  |  maybemodel = model parameters fitted to maybeinliers
7  |  alsoinliers = empty set for every point in data NOT in
   |  maybeinliers do
8  |  |  if point fits maybemodel with an error smaller than t then
9  |  |  |  add point to alsoinliers
10 |  |  end
11 |  |  else
12 |  |  |  continue
13 |  |  end
14 |  end
15 |  if the number of elements in alsoinliers is > d then
16 |  |  bettermodel = model parameters fitted to all points in
   |  |  maybeinliers and alsoinliers
17 |  |  thiserr = a measure of how well model fits these points
18 |  |  if thiserr < bresterr then
19 |  |  |  bestfit = bettermodel
20 |  |  |  besterr = thiserr
21 |  |  end
22 |  end
23 |  increment iterations
24 end
25 return bestfit
```

3.2.3 Fitness Function

The fitness of the line estimation with a subset of our data is measured by the standard linear correlation coefficient. Let S_X, S_Y, and S_{YY} be the

following:

$$S_X = X_1 + X_2 + \cdots + X_n \tag{3.1}$$

$$S_{XX} = X_1^2 + X_2^2 + \cdots + X_n^2 \tag{3.2}$$

$$S_{XY} = X_1Y_1 + X_2Y_2 + \cdots + X_nY_n \tag{3.3}$$

The fitness coefficient r, is given by:

$$r = \frac{nS_{XY} - S_X S_Y}{\sqrt{(nS_{XX} - S_X^2)(nS_{YY} - S_Y^2)}} \tag{3.4}$$

This coefficient is a measure of how well our data fits the current line estimation. Number close to 0 means little fitness and number close to 1 means good fit.

3.3 Implementation

The implementation of wall feature detection is embedded with the process scan line method (See SubVision.cc, void processScanline(...)), thus happens in parallel with the detection of features of other objects. The ransac wall fitting is called before vision sanity check. (See SubVision.cc pair$<$ *double*, *double* $>$ ransacWall()) In practice we allow maximum 10 estimation iterations, each with minimal of 8 data points. The fit of a line is described by the standard linear regression procedure[15]. The correlation coefficient is then used to measure such fit of a given estimated line on the data set. If such correlation is above 0.85 at a particular iteration for a line fit, we stop our program and return that line setting (A good fit is found). If after 10 iterations the best correlation is less than 0.55, we return NULL and claims that no field edge can be estimated otherwise we return the best line estimated among the 10 iterations.

The following figures demonstrate how much robustness this algorithm presents when running at only 10 iteration cutoff. 3.3, 3.4, 3.5, 3.6

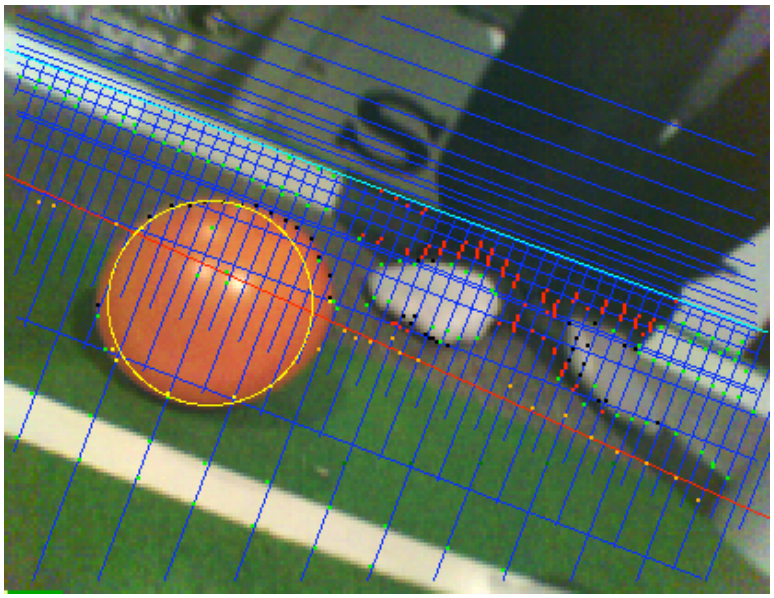

Figure 3.3: The filed edge is partially interrupted by the ball and the beacon at the right. 2006 Robocup World Championship, Bremen.

Figure 3.4: The filed edge in front of the blue goal been accurately marked. 2006 Robocup World Championship, Bremen.

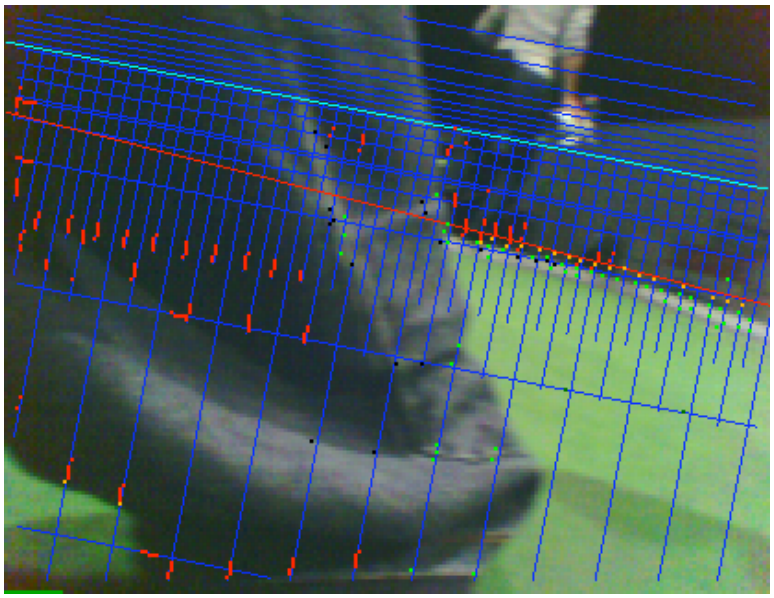

Figure 3.5: The ability to quickly eliminate outliers of point data when many of them are present. This image shows nearly half of the image are covered by an obstacle. 2006 Robocup World Championship, Bremen.

Figure 3.6: The same image presented at the beginning of this chapter. The field edge is marked by a thin red line elevated at the near half image height. This helps us to eliminate these complex background for ball and goal recognition. 2006 Robocup World Championship, Bremen.

3.4 Summary and Outlook

In this section we described our implementation of field edge detection. Being able to do so helps to sanity check out false object by their elevation. The procedure starts by recognizing wall features along each vertical scanline, where we are looking for green to non-green color transition but allowing some noise. At the next step we use a modified Ransac algorithm to prune those erroneous features and tries to come up with a wall line that agrees well with the rest of the wall feature set. The difficulty here is to deal with large number of inaccurate data points in our feature set. Sometime due to the short amount of time allowed, Ransac might also stop with non-optimal line fit even when few wrong points are contained in the initial domain. To overcome this, we might need to reduce the amount of false positive errors happened in wall feature recognition step. That is, we do not want to recognize a wrong wall feature (Be more conservative!). The false negative and positive error present an important trade-off in the balance between them. A perfect system is one such that these errors are perfectly balanced. In the domain of field edge detection, accurately identify every valid points and do not falsely mark any wrong points is very difficult.

Future work might also extent the system to handle recognizing multiple field edges when they present on one single image frame, for example, the corner of the field. Figure 3.7 shows a situation where two field lines meet at top right field corner and both should be marked in the future work.

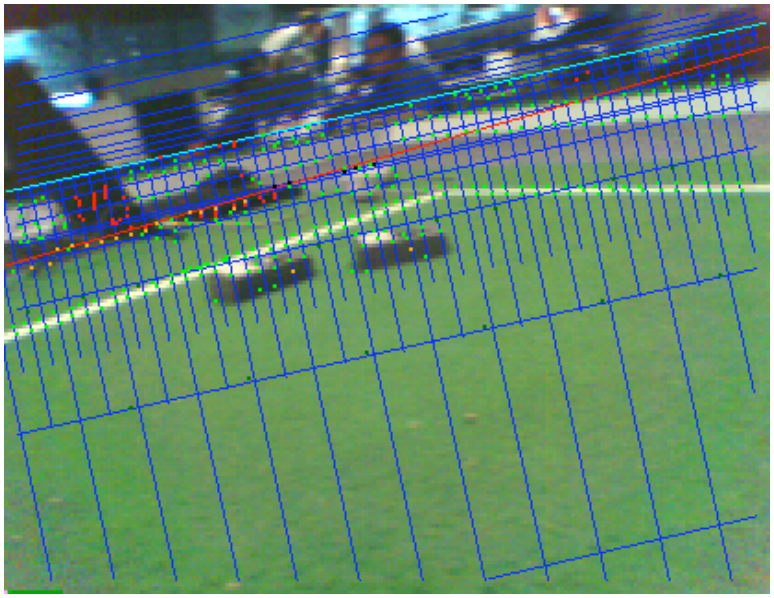

Figure 3.7: The present of multiple field edges needs to be recognized in the future work. Picture taken at 2006 Robocup World Championship, Bremen.

Chapter 4

High Level Behaviors

4.1 Behavior System Overview

The term "behavior" refers to the actions or reactions of a person or animal in response to external or internal stimuli. In the context here, we use the term "behavior" to refer to the way our robots process real-time information and make appropriate decision of responses. Behavior system is a sub-system in our software architecture that functions as such. The behaviors system gathers information from the sensors (Vision, Mechanical sensors), its memory maintained (localization) as well as information from its teammates (Wireless Transmissions). The output of the behavior system is usually a single action frame such as open mouth, raise tail, or an action in a sequence of actions such a group of actions that grabs a ball.

4.1.1 System Structure

Sitting at the top of behavior hierarchy is our most general modules - a forward player, or a goalie. The rule permits four players on the field including one goalie. Hence three of the four dogs will be running the forward player module and the remaining one will be running the goalie module. In real soccer, 11 players play different strategy and originate at different positions on a soccer field. Similarly, we assign "roles" to the three forward dogs. In general, one dog will be chasing the ball, manipulate, and tries to shoot into goal. Another dog prefers to place itself such that it is in support to the first dog on ball. The third dog usually sits at the back yard, getting ready to defense. These three characters are respectively called, "attacker", "striker", and "defender".

Unlike the FIFA world champion Italy where Del.Piero always places himself at the striker position regardless anything useful, we dynamically switch roles. That is, we assign the attacker role to the "best suited" dog,

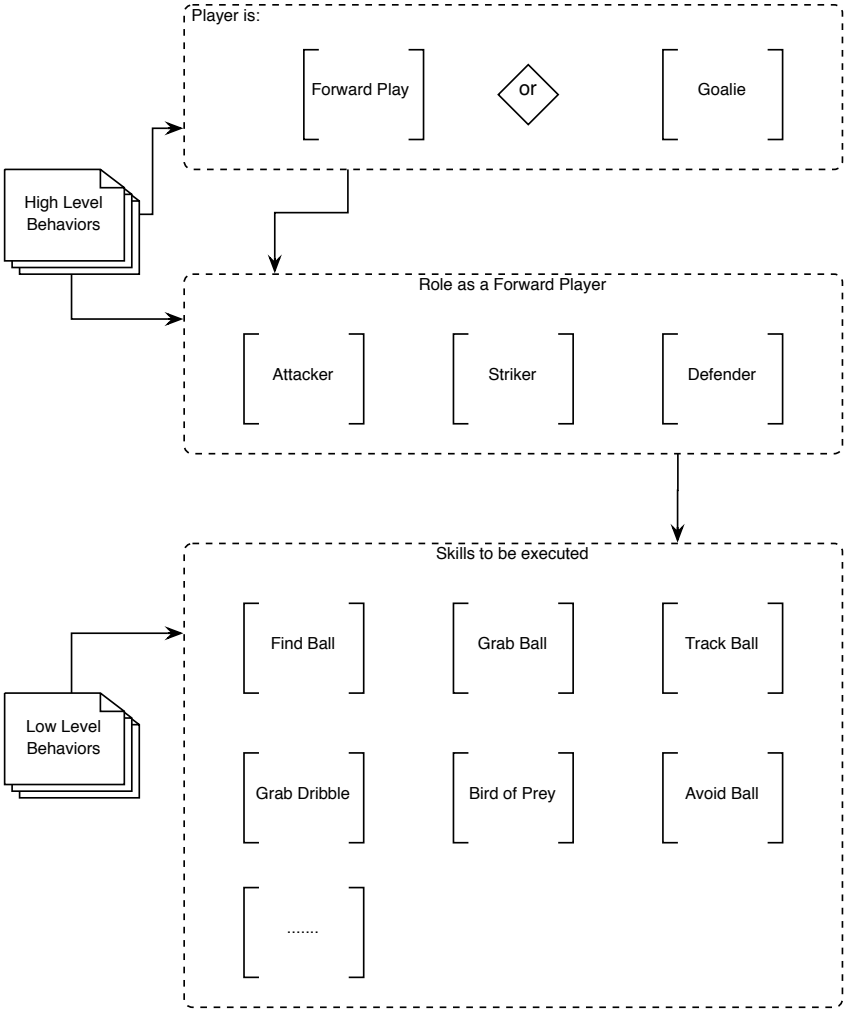

Figure 4.1: The rUNSWift behavior system structure. High level behavior is in charge of role, role positions and role assignment. Low level behaviors are those skills that can be used by various roles at high level modules. The ability to dynamically select which of these skill to execute by what role is known to be the "link" between the high and low level behavior modules. This link is implemented as a decision tree which is capable of generating decisions.

so does defender and striker roles. Dynamic role switching will be described in the later sections.

Having defined the characteristics of the three roles, one might find that strikers and defenders are relatively static comparing with attacker, because they are not chasing the ball all the time like the attacker. In fact both of these two roles have their so-called "home positions", which defines the expected position on the field under certain situation at particular time. Positioning module will also be discussed.

All three roles are capable to execute "skills". In particular, they are programmed to decide which skill to execute under what circumstance. The decision of which skill to execute is made based on the role type as well as other relevant information. A new decision can be valid even when actions of old decision have not yet completed. We will discuss how decision is made in the following sections.

Finally, a skill is a description/combination of a sequence of simple actions that complete a certain task, such as ball grabbing, ball finding, or ball dribbling. A complete run of a skill module spans cross multiple frames. A skill module is usually implemented by a state machine, knowing its current state, executes its next action. Important skill modules will also be explained.

Summing up, the behavior system consists logically: High Level Behaviors, which deals with who should be in what role and where their home positions are, A decision tree capable of making decision on what skill to run, and the Low Level Behavior modules that specify each basic skills.

The behavior system talks directly with Actuator Control, whose output signal is then received by the hardware through Open-R interface. See figure 4.2

4.2 Introduction

High-level behaviors mainly consist of role specification, role assignment and role positioning. Role specification deals with the definition of various roles at strategic level. For example, attacker focuses on the management of the ball, and shoots when appropriate. Defender on the other hand, might focus on how to place itself to the best defense position in a dynamic game environment. Moreover, role assignment is the job that decides which particular robot should be in which role at any time in a game. Finally, role positioning decides where to places each player on the field. This section

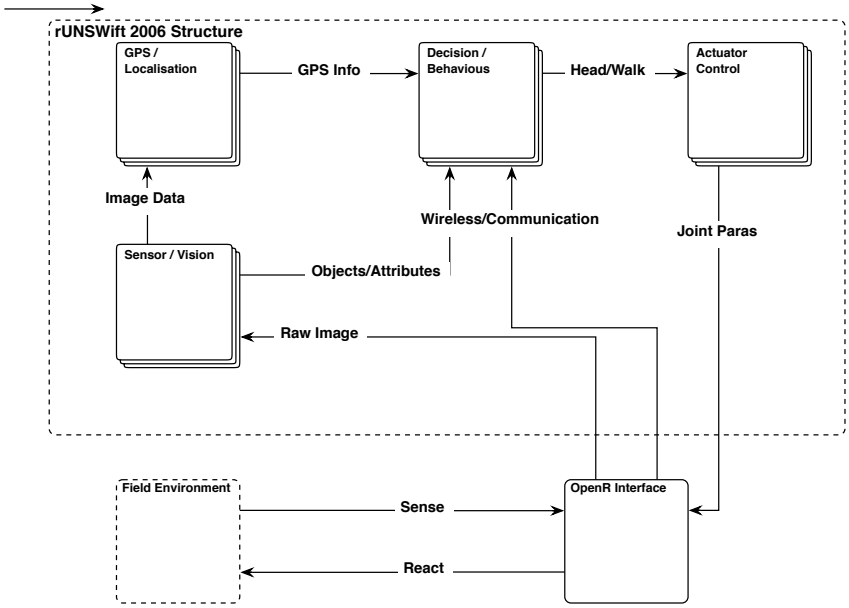

Figure 4.2: The rUNSWift architecture react with outside environment through OpenR interface. Agent at each frame sense and then react. Precisely, it first senses the world, store the results after processing image, update localization, make decision and finally set the joint parameters at the end of that frame.

will discusses the concept and implementation of these items in details.

4.3 Role

In this chapter, we will discuss the roles we designed - what is each role respond to. The roles used at rUNSWift are:

Role Type	Responsibility	Area in charge
Goalie	Defend the Goal	Only the goal box area, NEVER goes outside the goal box
Attacker	Track and chase the ball if the ball is not under control. Dribble and kick the ball if the ball is under control.	Attacker can be assigned to a robot at any where on the field.
Supporter	Stay close to the attacker. Play defence together with the defender if they do not have control of the ball. Be ready to receive the ball when team is "attacking"	Anywhere on the field
Striker	Similar to supporter, but typically keeps itself further away from the attacker when supporting comparing with a standard supporter.	Up half field typically
Defender	Defense the goal together with the goalie. Block incoming ball if it feels necessary.	Back third of the field

Table 4.1: The list of roles and their primary responsibility and active field area

4.3.1 Goalie

Goalie is an important role in a game. Experiment was conducted comparing an attacker shooting in to an empty goal with a goal defended by a goalie. The result shows that a goal with goalie is much harder to score than an empty goal. The following is a summary of goalie algorithm.

Goalie should always track the ball. In case of lost track of ball, goalie should quickly re-gain the visual ball. After tracking the ball for long time,

Algorithm 4.1: The typical layout of goalie logic

Input:
Output:
Result: Motion at each iteration of run

```
1  headScan() / headTrackBall()
2  x,y = BallX, BallY
3  if Facing the wrong way then
4  |  Direct body to the correct way
5  end
6  else if Found ball is coming towards the goal quickly then
7  |  Block the goal
8  end
9  else if Found ball is coming towards the goal then
10 |  Adjust stand position and heading according to the ball position
11 end
12 else if It is Ok to clear the ball by kicking it then
13 |  Select appropriate kick and kick
14 end
15 else if Ball is close to the goal then
16 |  Walk to the ball, and attack it
17 end
18 else
19 |  Select the best position and heading, and guard the goal
20 end
21 End of iteration
```

goalie also needs to look around for a short period of time to actively localize itself. These head motions are handled by the headScan() and headTrackBall() functions. To start with, goalie checks if it is facing up field. Sometimes goalie can get quite lost when it gets pushed so it is facing its own goal but believes it is actually facing the correct way but cannot see anything interesting. If this check returns false, goalie needs to turn around first of all. The second check is to see if ball is flying towards its own goal, for example, it gets kicked by the components. Goalie will do a full block if the ball is coming direction measured from goalie is small. Goalie will do a single direction block if that angle is biased to one direction. If the ball is coming but its velocity is small, goalie needs to quickly respond by moving its position and heading. Furthermore, if goalie detects that the ball is close to it and believes it is suitable to execute a kick to clear the ball but with caution that it will not score an own goal, then it will execute the kick. The goalie will then check if it is necessary to go forward and attack the ball. This happens when the ball is close to the goal, for example, on the edge on the goal box. The goalie would then walk to the ball and clear it by kicking it using one of its front arms. Finally, if goalie believes any of the above does not apply at this frame, it will pick up a "good" position and places itself there.

Goalie Positioning

Goalie positioning concerns about where to place the goalie in respond to the environment. People have come up with several policies that places goalie over years. Here we present one simple policy used at our Final Round with Nubots.

We draw one interval between the ball and the two goal posts. Call them respectively BR and BL (Ball− >Right GP and Ball− >Left GP). We then position the goalie's heading such that the line from goalie and the ball equally divides the angle BR-Ball-BL.

In addition, we also defined the vertical position of its neck position at (X, 20cm). Early work also uses variable Y component of such position. That is, if the ball is far away, then we increase the Y component to place the goalie further out from the goal. If the ball is close, we place the goalie on the goalie line. Thus, goalie is moving on a circular curve with radius Y.

The above goalie positioning is very simple to implement but performs very well in practice.

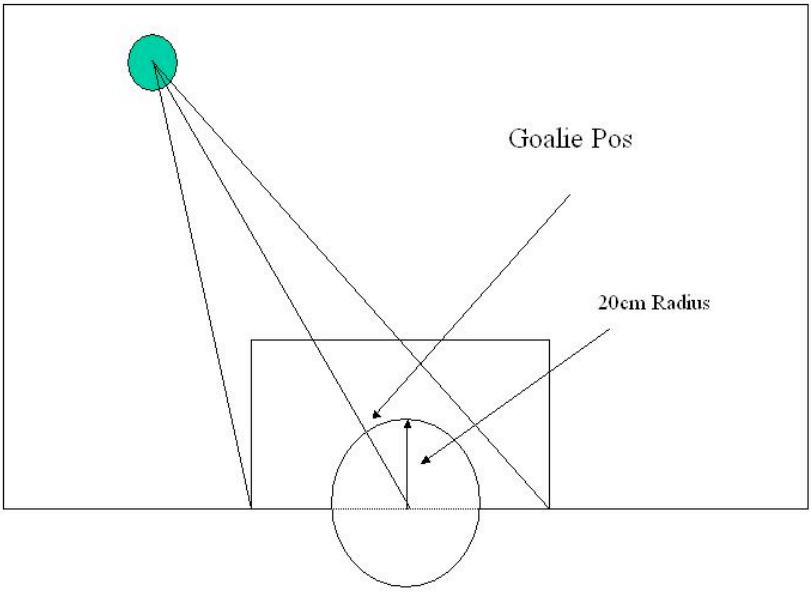

Figure 4.3: A simple way of placing of our goalie robot. This placement is easy to implement and works well in a competition.

Goalie Attacking

Goalie attacking is the behavior that goalie actively charging the ball and manipulate the ball appropriately afterwards. Goalie attacking in Robocup is not a new skill - Teams like Germany team[20],[41], WrightEagle[40], and NUbots[21] have all developed, and used this skill in a real game. The intuitive behind goalie attacking is that, rather than waiting in side the goal box, we want to get to the ball before the opponent and clear the ball hence avoids opponents on goal shots. This skill is useful when playing with team that often uses strong kick but requiring in possession of the ball before hands. i.e. GermanTeam and WrightEagle. A successful goalie attacking also gain the whole team time to backtrack the attacker and defenders into their positions.

rUNSWift 06 goalie walks to the ball and will execute U-Pen kick to either left-right directions. Interestingly, NUBots also allow their goalie to grab the ball. The advantage of possess the ball is that it allow the goalie to pick up the direction to kick the ball to, sort of like the real soccer. The disadvantages comparing with our go-kick simple approach is that it exposes them to the risk of missing the grab. Future work might extend the goalie to handle both "quick clear" and "ball control" when perform attacking. For example, if the opponent is very close to the ball or has equally the chance

to get to the ball first, we perform a quick clear of the ball without grabbing.

Goalie Scan

Goalie needs to track the ball if it can see it. In case goalie cannot see the ball, it needs to perform find ball. The find ball routine for goalie is vastly different from a normal forward player. In particular, goalie's primary find ball is achieved by scan its head around. This year the team has introduced a point head scan with variable duration for goalie rather than the traditional smooth scan routine used by other forwards as well. For example, these points might include left near beacon, left far beacon, right far beacon, right near beacon, close front and circular back to left near beacon. The intuitive behind this is its efficiency. One iteration is usually faster than a full scan. Secondly, it also helps to active localize it self, by looking at a direction a beacon is in. This work is primarily done by Oleg Sushkov in year 2006.

4.3.2 Attacker

When a robot gets assigned to be attacker, its main responsibility is to chase the ball and attack it in a good way. While attacker is doing these, it also needs to avoid going into its own goal box, actively localizing, and making decision on what exactly low level skills to execute. More details will be present in the later chapters. Before we dig into it, algorithm 4.2 summarizes the property of a typical attacker's run.

Own Goal Box Avoidance

If a robot other than goalie goes in to its own goal box, it will be panelized and taken out from the field for 30 seconds. This is not desired. Due to the nature of the attacker, we need these extra lines of code that makes sure it does not violate this rule. Other roles such as defender and supporter have their position defined for them, so they do not need to worry about goal box avoidance.

This year we use the following criteria to trigger goal box avoidance:

- The ball is in our own goal box, AND
- We have not grabbed the ball, AND
- We have not started to grab

Algorithm 4.2: The typical layout of attacker logic

```
Input:
Output:
Result: Motion at each iteration of run
if Ball in goal box then
    AvoidOwnBox()
end
else if Need to active localise then
    ActiveLocalise()
end
else if Need to kick/grab ball then
    Select what exactly to do, and execute the selected strategy
end
else
    Chase the ball, track the ball
end
End of iteration
```

The goal box avoidance is then accomplished by the following algorithm:

Algorithm 4.3: Perform own goal voidance

```
Input:
Output:
Result: Place attacker at the best position when avoid own goal
if Ball still far away then
    return (Do Nothing)
end
else if We are on either sie of the goal box then
    Walk to the nearest goal box corner
end
else
    //here we must be in front of goal box
    Walk to the side of the ball
    //so goalie can effectively clear the ball
end
End of iteration
```

Active Localisation

After tracking the ball for long time, robots may get worse precision of its position. In the context here attacker may get lost about where it is and consequently make wrong strategy decision. Similar issues are present for other players such as goalie, defender and striker. To overcome this, one may want the dog to look at landmark for a short period of time and then back on tracking the ball as usual. This skill is then repeated from time to time.

For attacker, active localization is triggered by:

- The attacker has just span to find the ball.
- The attacker has seen the ball for a few frames.
- The ball is still far away and straight ahead.
- sFindBall.lastCloseFrame is not recent.
- sGrab.gLastApproachFrame is not recent.
- The attacker has not recently actively localized.

4.3.3 Supporter / Striker

Supporter's main responsibility is to place itself in a good position such that it is in support with the attacker when attacking, and is in support with defender when defending. Striker is similar to supporter. The difference is that striker tends to stay at the front half of the field more. Even the team is doing defense, we still want to program the striker to sit at the front to receive up coming ball. These difference can be controlled by various specification of their home positions. The main logic run on Supporter/Striker can be summarized in 4.4:

Stuck Detection

The stuck detection/handling concept was originally developed by 2001 team using visual detection.[11][17] It allows robot to detect if it is in a situation

Algorithm 4.4: Summary of the run of Supporter/Striker

Input:
Output:
Result: The run of Supporter/Striker at each iteration

```
Track/Find the ball
if Stuck forward then
    Return
end
if Any teammate has grabbed the ball then
    Get out of the way
end
else if Need to avoid ball currently being handled by teammate then
    Avoid the ball
end
else if Bird of Prey is already runing then
    Bird of Prey
end
else if Obstrcting attacker then
    Avoid attacker
end
else if Too far from home position then
    Bird of Prey
end
else if Lost the ball then
    Find ball
end
else
    Move back to home position
end
End of iteration
```

where there are obstacles closely surround by. For example, obstacle might be opponent robot, goal post, beacons, and field wall if there is. Being able to detect stuck into such objects are very useful.

This year we mainly used paw sensors, sensors sitting at the front of both legs. Our walk does not interfere with these sensors at all, and they only trigger when they hit something. If non-attacker robot walks into an object, for example a beacon, it will then walk backwards for certain amount of time and goes on from there. In particular, if both left and right sensors triggered, a robot would walk straight backwards. If only left leg detected obstacle, a robot would walk backward as well as to the right. Same idea apply to the situation where right leg detects obstacles.

Similar to 2005 year, this concept is also applied to ready state. A ready state is a 45 second time duration in which every robot on the field should go back to their initial game beginning positions. While robots are walking towards each other side of the field, stuck detection helps the robot on our team to back track and avoid the up coming robot.

Avoid attacker

Avoid attacker is essentially another form of backing off, similar to stuck state handling. However, their intuitive are quite different. Loosely speaking, avoid attacker means get out of the way of the attacking route of the current attacker on the same team.

The original idea of this behavior came from rUNSWift year 2002,[18] and polished by the 2005 and 2006 teams. There are two ways of avoidance designed. The first one caters for attacker with ball in possession, hence non-attacker interfere with the attacking route should make itself clear. See figure 4.4

Another form of avoidance happens when attacker has not yet gained the ball but is on its way for the ball. Here we also would like the non-attacker to clear itself from the attacker. See figure 4.5

Bird of Prey

In year 2003, rUNSWift developed a behavior to counter the lack to defensiveness by efficiently. [19] This behavior was then named bird of prey by the 2003 rUNSWifters. Through the development over years, Bird of Prey this year is used to allows non-attackers to walk to their home positions on

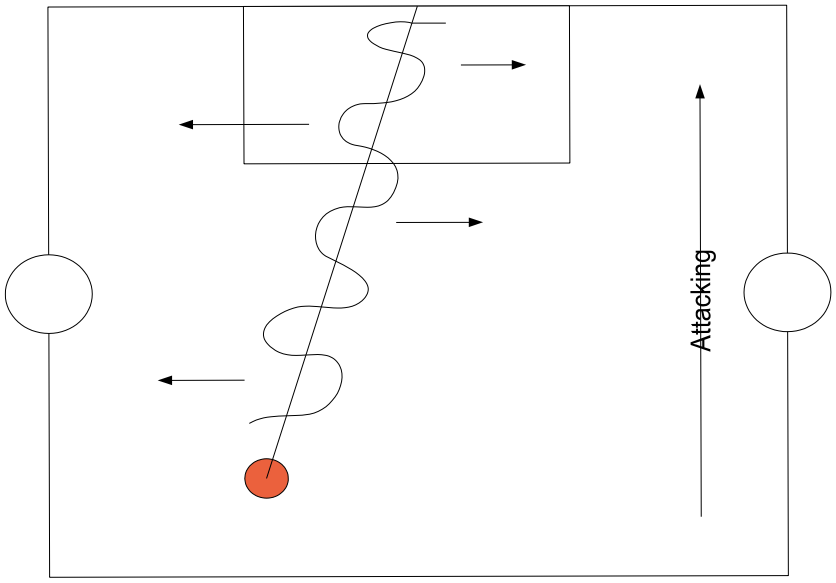

Figure 4.4: Attacker avoidance plan A. Non-attacker should avoid being close to the line between the ball and the goal

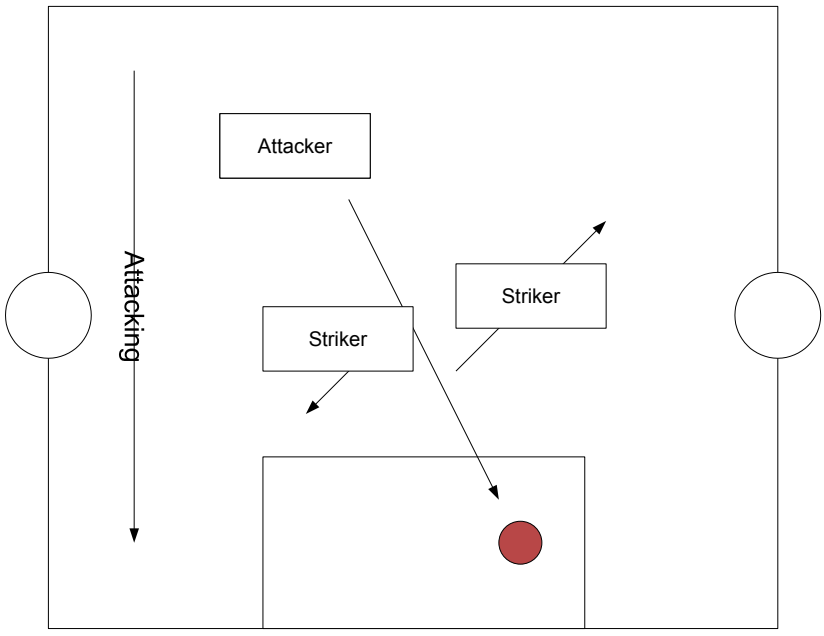

Figure 4.5: Attacker avoidance plan B. Non-attacker should also avoid being interfere with an attacker when attacker is chasing the ball

the field via a safe path. And does not get obstructed by objects along its journey. Along with its trip, we also use stuck avoidance to detect and avoid obstacles.

In a dynamic role switching environment, a non-attacker might suddenly realize it is out of its home position by a significant amount, and there are teammates and opponent robots on the routes that directly links where the non-attacker is and its desired home position. Bird of Prey is used here to guide the robot to go around these obstacles and quickly return to its home position. See figure 4.6

Figure 4.6: Bird of Prey in Action. Picture taken from a practical match between rUNSWift(Red) and GermanTeam06(Blue) From TopLeft to BottomRight, Red 1 is Goalie. Red 2 is Attacker chasing the ball. Red 3 is Defender doing Bird of Prey. Red 4 is the Striker at the front. We see Red 3 is walking in a circular fashion and tries to avoid slowing down due to going home directly through teammate and opponent robots.

The 2006 Bird of Prey is successfully implmented in module sBirdOfPrey.py in PyCode directory.

4.3.4 Defender

The main role responsibility of defender is to stay at the back of the field and wait for offensive opportunity. Unlike supporter, striker and supporter, defender is only active within a fix area at the back. Positioning will be discussed in the later chapter. The run of defender logic at each iteration is similar to supporter and striker.

4.4 Role Switching

Having defined which roles we have and what does each role do, this section describes our role assignment mechanism - Dynamic role switching. In a game we assign roles to the four players on the field. Since no player other than goalie is allowed in the own goal box, one robot is always assigned to be a goalie. For the rest of the three players we dynamically assign most appropriate role to each of them at any time in a game.

Dynamic role switching is advantageous that the robot in the most appropriate situation gets assigned with the according role. The challenge here is that we also need to tune up well the system so they play coherently in a game. For example, we do not want to see all three robots get assigned to be defenders and no one is willing to go for the ball. Neither do we want to see more than one robot are in attacker role so they fight against each other while on their way to the ball together.

For each dog, we use the following set of algorithms to determine its role.

4.4.1 Attacker Selection

Each time we ask what role should I be in, we first check to see if we can be an attacker. If there is no attacker in the team, then be attacker. If there is already an attacker, but for some reason that attacker can not see the ball and I am currently close to the ball and can see it, then I will be attacker. If I am about to grab the ball and currently doing behaviors with the ball, then continue to be attacker. If none of the above holds, I calculate my and teammate's time to reach the ball. If I have the shortest time to get to the ball, then I will become attacker. See figure 4.6

Time to reach ball

Time to reach ball is the measure of how fast can a robot get to the ball. We prefer robot that is able to get to the ball in the shortest time to under-

Algorithm 4.5: Top level role switching. We first consider if a robot should be attacker. If this robot should not be attacker, we then consider defender.

Input:
Output: The role the robot should be in
Result:

```
if Should I be Attacker then
    Become Attacker
end
if Should I be Defender then
    Become Defender
end
else
    Be Striker
end
End of iteration
```

Algorithm 4.6: Attacker selection requirements

Input:
Output: If the robot can be an attacker
Result:

```
numAttacker = countAttacker()
if numAttacker == 0 then
    return True
end
if numAttacker > 0 AND canSeeBall AND AmClosestToBall then
    return True
end
isGrabbed = HaveIGrabbedTheBall()
grabbingCount = HaveIStartedToGrab()
if isGrabbed OR grabbingCount > 0 then
    return True
end
myEstimateTime = timeToReachBall()
while RobotI in Other Robots do
    if myEstimateTime > RobotI.timeToReachBall() then
        return False
    end
end
tie = BreakTie()
return True AND tie
```

take attacker role. This measurement result is calculated for each robots on the team except the goalie, and shared through wireless LAN for individual robot's role calculation.

The time to reach ball concerns about many attribute of the robot. For example, the straight-line distance from the robot to the ball might be a reasonable indication. We might also account for a small amount of delay if the robot is facing opposite to the ball direction so it needs time to turn and walk. In addition, there might be obstacles, such as team mate and other opponents on the way, this might add even more magnitude of delay.

In fact, the concept of combining relevant information and formulate the approximation of time for a robot to reach ball is not new. Back to year 2004, German Team first time described their formula on their technical report. [20]

$$\begin{gathered} timeToReachBall = distanceToBall/0.2 \\ +400.0 * fabs(angleBetweenBallAndOpponentGoal) \\ +2.0 * timeSinceBallWasSeenLast \end{gathered}$$

The above formular states the time to reach to the ball for a robot is the weighted sum of its dynamic attributes. In particular, these attributes are:

- Distance to the ball
- Angle between ball and the opponent goal
- Time since last time seen the ball

At rUNSWift, we have adopted this approach since 2005. The current approach consists of mainly two steps. First we calculate the time required to reach the ball position, a point. Next we calculate so-called "bonus" time under the current situation. For example, robot one might have the lowest ball variance, hence we might want to bias towards to this robot slightly. Take another example, we might want to add some hysterisis on the current attacker, so we assign bonus on the current attacker so it is more likely to stay on the same role. The idea of "bonus" is exactly to capture "preference" as such, subject to flexible modifications. The final time to reach ball is then the sum of these two.

At step one, we calculate the standard time to reach an arbitrary point on the field, here the point being the ball. We first calculate the straight-line

time to get to the point. We derive the result by simply divide the distance by the estimated velocity. We then increase the time if we need to turn first in order to facing to the correct direction. We also increase the time if we realize there are obstacles between the routes. The obstacles are calculated from the obstacle map stored, and gets updated every frame.[1] In addition, we also perform stuck detection and increase the time upon the found of stuck stage.

At the second step, we add in our "preferences". The typical bonus we used at 2006 are:

- Certainty of where the ball is
- Current attacker hysterisis
- Grabing bonus

The certainty of the ball position can be measured from the variance of the ball - bigger bonus for tighter variance. Alternatively we can count the frame of lost of the ball. The bigger the less the bonus will be. Typically we apply negative quadratic bonus with respect to the lost ball frame. With current attacker hysterisis, we add bonus if this robot was an attacker at last frame. Last but not least, we give big bonus to a robot that is about to grab or is grabbing the ball.

4.4.2 Defender & Striker Selection

The algorithms for defender and striker selection are similar to the approach we discuessed for attacker. Apart from the attacker we just selected in the above procedure, the robot that has shorter time to reach Defender's "home position" will be selected to be the Defender. The last robot will be selected to be the Striker.

The entire role switching ststem is successfully implmented in module pForward.py, which consists of the top-level role query calls. Time to reach ball and "bonus" concepts are subsequently implmented in hTrack.py module.

[1] For a detailed description of obstacle map please refer to rUNSWift05 Thesis Report. In particular, Real-Time Shared Obstacle Probability Grid Mapping and Avoidance for Mobile Swarms. [22]

4.5 Role Positioning

4.5.1 Background

In the previous sections, we described our roles and role switching system. This section will introduce the positioning system that controls where to place robots in a dynamic fashion. Strictly speaking, the positioning system concerns only about defenders, supporters and strikers.[2] This section would focus on our strategy of placing these non-attackers.

The traditional approach is usually come up with some form of hand-coded decision tree before hand. Then deriving the position to place the robot, we traverse the tree with a vectors of parameters that specify the current game situation, and the output is a tuple of X-Coordinate, Y-Coordinate. We will start by introducing several version of this approach that we have used in year 2006.

4.5.2 Decision tree based positioning

Striker Position

The striker position is determined upon where the ball is. We divide the field into three zones. We have different policy for each zone if the ball is in it. The first zone spans from the field length to field length times 0.6. (540cm - 325cm). The second zone starts from half of the field length and ends at 325cm. The bottom half field belongs to the third zone. Recall that the main responsibility of striker is to support attacker and stay at the front. So we do not want to place it too far back to interfere much with defense. The striker position policy is summarized as in figure 4.7.

Supporter Position

We have developed several position strategies for our supporter player. One class of the policy uses the same zone dependent policy specification approach, where we divide the filed into smaller pieces, more than striker position. We then specify our policy for each of these small pieces of field.

The second fashion of supporter position we found very effective is to position the supporter at the back of our attacker. This approach is trivial to implement but very active in practice. In many cases our supporter can quickly catch up the ball and back up the attacker, in case the attacker gets

[2] Attacker has a flexible positioning, since it chases the ball. Goalie determines its own position.

Algorithm 4.7: The striker positioning algorithm

Input:
Output: Striker's position
Result: targetX,targetY

```
1  yOffset = 80.0
2  ballX,ballY = ballPosition()
3  if ballX > FieldWidth*0.5 then
4  |  targetX = Striker_Left_X
5  end
6  else
7  |  targetX = Striker_Right_X
8  end
9  if ballY < FieldLength * 0.5 then
10 |  targetY = FieldLength * 0.5 + yOffset
11 end
12 else if ballY < FieldLength * 2/3 then
13 |  targetYA = ballY - yOffset * 0.65
14 |  targetYB = FieldLength * 0.5
15 |  targetY = max(targetYA, targetYB)
16 end
17 else
18 |  targetYA = ballY - yOffset
19 |  targetYB = FieldLength * 0.5 - yOffset
20 |  targetY = min(targetYA, targetYB)
21 end
22 return targetX, targetY
```

lost or gets stuck with opponent. See figure 4.8

Algorithm 4.8: The supporter positioning algorithm. Supporter is a similar role comparing with striker but supporter also helps defense.

Input:
Output: Supporter's position
Result: targetX,targetY

```
1 yOffset = 80.0
2 xOffset = 70.0
3 ballX,ballY = ballPosition()
4 targetX = min(max(ballX, 70.0), FieldWidth - xOffset)
5 targetY = max(ballY - yOffset, 120.0)
6 return targetX, targetY
```

Defender Position

Defender's vertical component Y is kept at 120cm behind the ball, but must be at least 20 cm away from the top of the goal box. Its horizontal component X is controlled by the magnitude of the horizontal displacement of the ball. The bigger the displacement of the ball is, the more biased to one side we place our defender on the horizontal direction. Finally, we have a simple special case when the ball is very far forward in opponent half, we place defender behind the drop in point. See figure 4.9

Implementation

Positioning system is successfully implmented in hWhere.py, through getDefenderPos(), getStrikerPos() and getSupporterPos().

4.5.3 Weighted Regression based Positioning

Introduction

In year 2006 we decided to use a direct representation of a reactive strategy as a function from inputs to outputs. We have a vector of input variables used to decide out current strategy, and we have a vector of output variables that are selected by the strategy code. The strategy itself is a function between these two vectors. The declarative function representation is data driven. We read the training set into memory, and we delay the processes of the queries until it needs to be answered. Within the context of positioning, a set of expert-specified input-output vectors is stored. Each input vector

Algorithm 4.9: The striker positioning algorithm

```
   Input:
   Output: Defender's position
   Result: targetX,targetY
 1 ballX,ballY = ballPosition()
 2 targetY = max(GoalBoxDepth + 20.0, ballY - 120.0)
 3 if ballX > FieldWidth * 0.5 then
 4 |   fraction = (ballX - FieldWidth * 0.5) / (FieldWidth * 0.5)
 5 |   targetX = FieldWidth * 0.5 + 60.0 * fraction
 6 end
 7 else
 8 |   fraction = (FieldWidth * 0.5 - ballX) / (FieldWidth * 0.5)
 9 |   targetX = FieldWidth * 0.5 - 60.0 * fraction
10 end
11 if ballY ¿ 300.0 then
12 |   if ballX < FieldWidth * 0.35 then
13 |   |   targetX = 50
14 |   end
15 |   else if ballX ¿ FieldWidth * (1 - 0.35) then
16 |   |   targetX = FieldWidth - 50
17 |   end
18 |   else
19 |   |   targetX = FieldWidth + (ballX - FieldWidth * 0.5) * 0.5
20 |   end
21 end
```

describes a game situation. While each output vector contains the corresponding preferred position and heading of our robots. Answer a preferred position query requires finding the relevant vectors from the training set and finally the relevance is measured by some form of distance functions. In the following sections we will describe this new approach in depth.

Concept Overview

The position and heading of the two non-attackers compose the out put vector V_{out} consisting of X, Y, Heading for the defender and the striker. The input vector V_{in} is a tuple of four values, ballX, ballY, attacker_angle_to_ball, and Attcking_Defencing. The last variable is a boolean variable that is true when team is attacking and false otherwise. Semantically, we would like to use ball's position on the field, attacker's relative angle to the ball, and a 1 bit attacking or defending variable to conceptualize the current game state.

To build training set, we come up with entries that maps from random input vectors to the most appropriate output vector defined by human expert. The bellow strategy graph shows some example entries. 4.7

To answer a query given a input vector V_X, we first find all entries in the entire training sample that are relevant to V_X. Relevance here is controlled by the distance function between V_X and any V_i in our training set. The output vector V_Y is then computed from the output vectors these relevant entries.

Distance Weighted Averaging

We first consider the naive case, distance weighted average. Given $\hat{x}$, and set of entries $[x_1 \rightarrow y_1, x_2 \rightarrow y_2, \ldots, y_n \rightarrow y_n]$, we can predict $\hat{y}$ by:

$$\hat{y} = \frac{\Sigma y_i}{n} \tag{4.1}$$

Here we take the average on Y_i in $[Y_1, Y_2, \ldots, Y_n]$. If all of these values are relevant then the above formula would make positive sense. Conversely if few of these values are in fact relevant to the value we are querying for, then the above formula makes little sense. We can solve this barrier through two ways that are essentially equivalent: Weighting the data directly or weighting the error criterion used to predict the value.
[23]

As for the first approach, we weight the data directly according to their distance from the query point. We use dis(x, $\hat{x}$) to calculate the distance

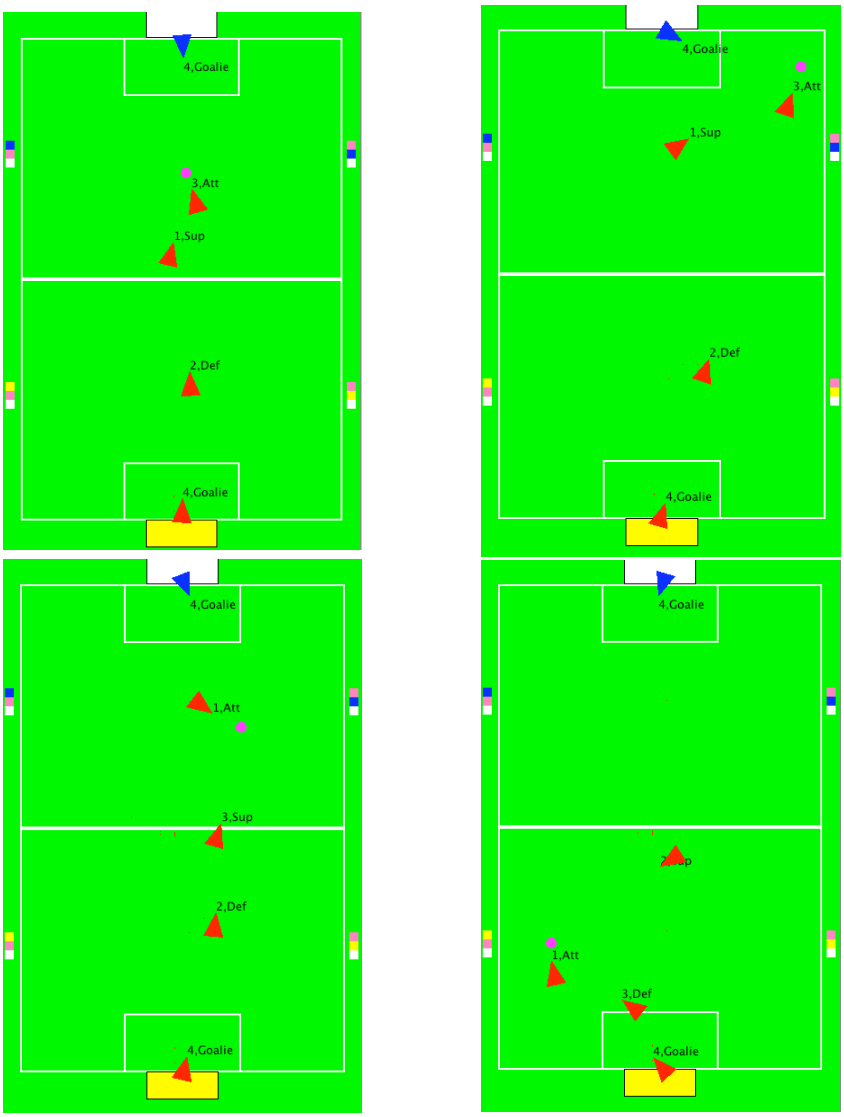

Figure 4.7: Non-Attackers positions specified by human under various game conditions. TopLeft: The ball is in opponent half field, we place defender and supporter directly behind attacker. TopRight: The ball is at the top right opponent half, we place our defender at the back but slightly more forward than the TopLeft situation. We also place our supporter at the center near opponent's goal box, and head its direction towards to the ball. BottomLeft: Here we see our attacker is chasing the ball and facing towards our own goal, we will thus place defensive strategy rather than offensive attacking, and we place our supporter near the center circle to form team's first line of defense. BottomRight: The ball is at the bottom left region. We place our supporter at high elevation to wait offensive chance and be ready to receive ball passed forward by the attacker.

between a data point (x, y) and query point ($\hat{x}$, $\hat{y}$). The distance function we used here is the Euclidean distance:

$$dis(x, \hat{x}) = \sqrt{\Sigma(x_i - \hat{x}_i)^2} \tag{4.2}$$

The weighting function W() maps Euclidean distance to the weighting factor is introduced. The predicted $\hat{y}$ is now the then the ratio between the weighted sum over the sum of the weights:

$$\hat{y} = \frac{\Sigma y_i W(dis(x_i, \hat{x}_i))}{\Sigma W(dis(x_i, \hat{x}_i))} \tag{4.3}$$

As for the second approach, we now weight the error function. Intuitively, we want to fit nearby points well, but we care relatively less about such "fitness" for points that are far way. Christopher, Andrew and Stefan formulated this idea as:

$$C(q) = \Sigma(\hat{y} - y_i)^2 W(dis(x_i, \hat{x}_i)) \tag{4.4}$$

Locally Weighted Linear Regression

Locally weighted regression (LWR) generates query result that fit locally well within our training set. In the context of our positioning, the estimated positions should fit well with those entries in our training set that are of similar game situations. By similar, again we mean the distance function defined return small magnetite between x_i of one entry, and $\hat{x}$ which is the query point. We will discuss the standard approach as described by Atkeson. [23]

To start with a global linear model of parameter β can be expressed by:[24]

$$x^T \beta = y \tag{4.5}$$

We then append constant 1 at after all input vector x_i. Hence we have training set:

$$X\beta = Y \tag{4.6}$$

Using unweighted regression, we can then go on to approximate β to minimize criterion:

$$Err = \Sigma(x^T \beta - y_i)^2 \tag{4.7}$$

Apply direct data weighting approach as set up, we then calculate the distance from each stored points in our training set to the query point $\hat{x}$.

The weight for each such point is then obtained by:

$$w_i = W(dis(x_i, \hat{x}_i)) \tag{4.8}$$

where W is our weighting function.

We then apply this weight with its corresponding row in our matrix X. This involves multiply it with x_i as well as y_i, let us use Z to express the left matrix after multiplication and V to express the right matrix after multiplication, this is:

$$X\beta = Y$$
$$Z = WX$$
$$V = WY$$

Solving the normal equations:

$$(Z^T Z)\beta = Z^T V \tag{4.9}$$

And this gives us estimate for $\hat{y}$:

$$\hat{y}(\hat{x}) = X^T (Z^T Z)^{-1} Z^T V \tag{4.10}$$

Some source also point out that inverting $(Z^T Z)$ might not be the most efficient approach to solve the equations. [25]

Kernal Functions

Kernal function, also called weight function, maps distance from the query point to a weight ratio. The maximum weight should occur at distance zero and decay smoothly as we span across our domain.[26]

The kernal function we used is the smoothing weight function: Gaussian Kernel. [27][28][37]

$$W(d) = \exp(-d^2) \tag{4.11}$$

This kernal function has infinate domain and we can easily adjust a shreshold value to ignore data further from a particular radius from the query. [23]

4.5.4 Summary and Future Work

Retrieving Relevant Data

Application of locally weighted regression on relatively large training sample, for example, more than 1000 situation → positions entry set can be time consuming. This brings us the concern of the speed of algorithm. If we do a full weighted regression on every entries in the set, this requires a O(n) lookup and a massive matrix manipulations overheads. Hence people tend to apply weighted regression "Locally", only on entries that seem to be relevant. Given a distance function dis(x_i, $\hat{x}$), there is fast algorithm that computes the nearest points with the lowest dis magnitude. In addition, this approach allow us to find the nearest N points with the lowest distance magnitude. This therefore made it feasible to apply locally weighted regression with faster retrieving speed.

The trick we play here is to store our training set into a k-d tree structure. Finding nearest n nodes between a fix node x in a naive structure like an array takes linear time. Nearest neighbor in a k-d structure performs asymptotically logarithmic in the above situation. See figure 4.8

A dk-tree is a space-partitioning data structure for organizing points in a k-dimensional space. [30],[31],[32],[33] To build a static kd-tree from n entries from our training set takes O(n log(n)) time. And the nearest N nodes lookup takes O(log(n)) time to complete. This accelerates our search speed so relevant data entries can be quickly selected for higher up regression calculations.

Conclusion

Regression positioning is an application of in-memory lazy learning, where we read the entire training set into memory, stored them into a data structure for efficient look up, and delays the process until a query needs to be answered. Unlike the decision tree based positioning approach, here we allow human expert to specify the desired positions and heading for non-attackers directly, under a static game environment on a frame-by-frame basis. Experiment in the lab shows this approach can already perform as well as our traditional approach by using less than 50 entries. To come up with a comprehensive training set, we have also allowed symmetricity, that is, we only define situations on one half of the field (usually left and right along the field width), and the corresponding situation happened on the other half is calculated symmetrically. Secondly, a good concept of "locality" may need to be further investigated in the future. The distance function matrix is vital here because it specify data's relevance directly. Apart from

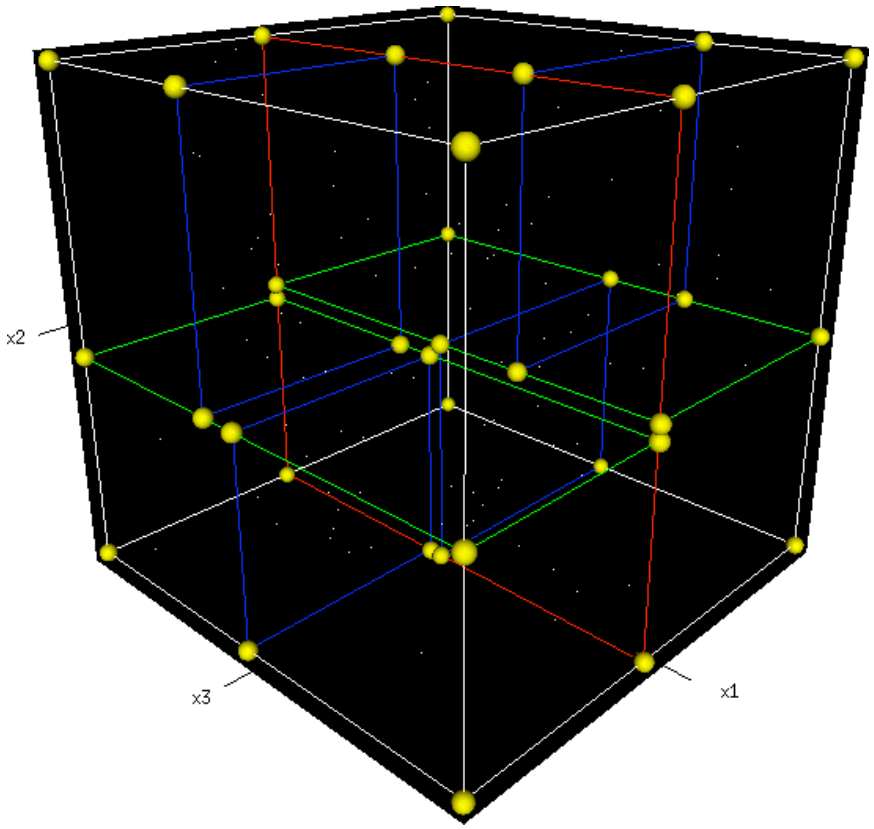

Figure 4.8: A 3-dimentional kd-tree.[29] The first split(red) cuts the root cell (white) into two subcells, each of which is then split(green) into two subcells. Each of those four is split(blue) into two subcells. This leaves us with final eight leaves.

simple Unweighted Eculidean distance used as above, one can also take a look at "Fully Weighted Euclidean distance", also known as Mahalanobis distance,[34], Minkowski norm, or function that has various weight on the input vector.

At rUNSWift 2006, both decision-tree and a naive version locally weighted regression have been implemented for our positioning system. And we can easily switch between one another for our positioning. However, decision-tree based approach was used in the world final competition. Future work on the regression based positioning will go on.

Chapter 5

Generating Decision

5.1 Introduction

In previous chapter we discussed our behaviour system architecture, which consists of high-level behaviour layer and low level behaviors layer, and a player picks up the most appropriate low level behaviour module to execute through our decision making system.

In this chapter we would like to describe our decision-making system. How does a player make decision on what to do next, especially attacker player.

5.2 Using Decision Tree

Using decision tree to generate appropriate action to execute is not recent at rUNSWift. A decision tree takes input a situation described by a set of attributes and returns a decision, the predicted output value for the input. A decision tree reaches its leaves by performing sequence of tests. An internal node represents a test, and a leaf node represent an output.

The game situation is described by many parameters. In fact, there are too many to take into account. Thus we only consider a small subset of them - the attributes that we feel important and fundamental in modeling a game situation, and could well affect what action an attacker should take in the next step. These attributes can include:

- The position and heading of the team, including the goalie
- The ball position and velocity

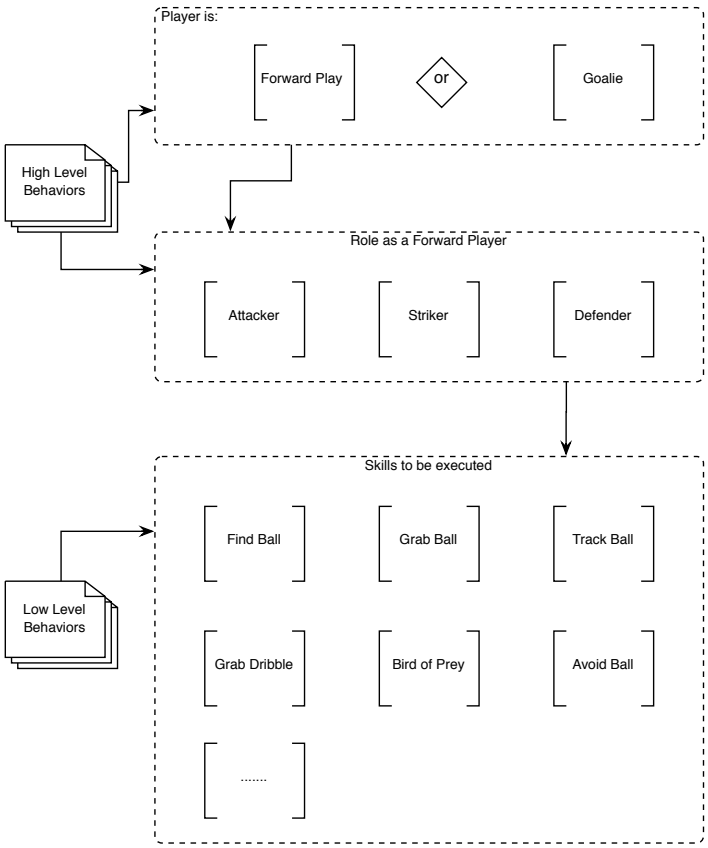

Figure 5.1: The rUNSWift behavior system structure. here decision system is represented bt the linkage between high and low level system.

- Defensive or offensive strategy playing
- Who has the control of the ball
- Opponent players' positions and headings

In general, we add in those attributes that we feel could affect the next-step decision-making. For example, we might add in the score difference into the decision tree used by our goalie, hence to implement a goalie that goes to the front and become an attacker when the team is down by more than 3 scores.

Finally, the decision made on leaves contains two classes of actions. The first is a class of actions that does not require grabbing the ball. This class of actions can quickly clear the ball using various part of the body with minimal time delay possible. The disadvantages of these actions are their low accuracy and reliability. The second class of actions require grab the

ball first, then actions with ball handling are followed. The advantages are the good control of the ball with high reliability. The main disadvantage is the grabbing ball it self adds time overhead delay.

5.3 Field Zones

At the top of our decision tree, we query the ball position attributes. And this test channels our query into 3 branches. They are, OffensiveThird, MiddleThird, and DefensiveThird. See figure 5.1

The decisions made on each field would have different emphasizes. The handling of the ball in defensive area may focuses on speed, that is, chasing the ball and clear it as quickly as possible, the requirement of the ball trajectory post-action is often somewhat less important. Loosely speaking, given the choice between slowly approaching the ball and letting the opponent acquires the ball and clearly the ball quickly even with the chance of getting it off the field, we will go for the later one. The strategy we want to emphasize is to minimize the grab of the ball from our opponent in front of our goal.

When we are near the field middle, we would still prefer our quick response, i.e being able to quickly react reasonably, but we also require some form of accuracy. Unlike defensive field third, we start to hate knocking the ball off the field within this area. The strategy we play at the middle can be summarized by two words: speed and accuracy. We want to quickly move the ball along and push it up field. Finally, when we are at the front third field, accuracy starts playing the top priority. Actions to shoot and avoid obstacles at the front need to be accomplished with high quality.

Algorithm 5.1: Top level decision tree divides field into zones and channel the query down upon which zone we are playing in.

```
   Data:
   Result:
1  ballX,ballY = getBallPosition()
2  if ballY < FIELD_LENGTH * 0.4 then
3  |  return SelectInDefensiveThird()
4  else if ballY < FIELD_LENGTH * 0.6 then
5  |  return SelectInMiddleThird()
6  else
7  |  return SelectInOffensiveThird()
8  end
```

Within each of these zones, there are sub-zones. Within defensive and

middle field thirds, we have left edge slide and right edge slide, as well as middle slide. So we can react differently when we are near the edge. The offensive third is further split into six logical regions. They are, top left corner, top right corner, left edge, right edge, opponent goal close region and the rectangular rest area. This further break is figured as 5.2:

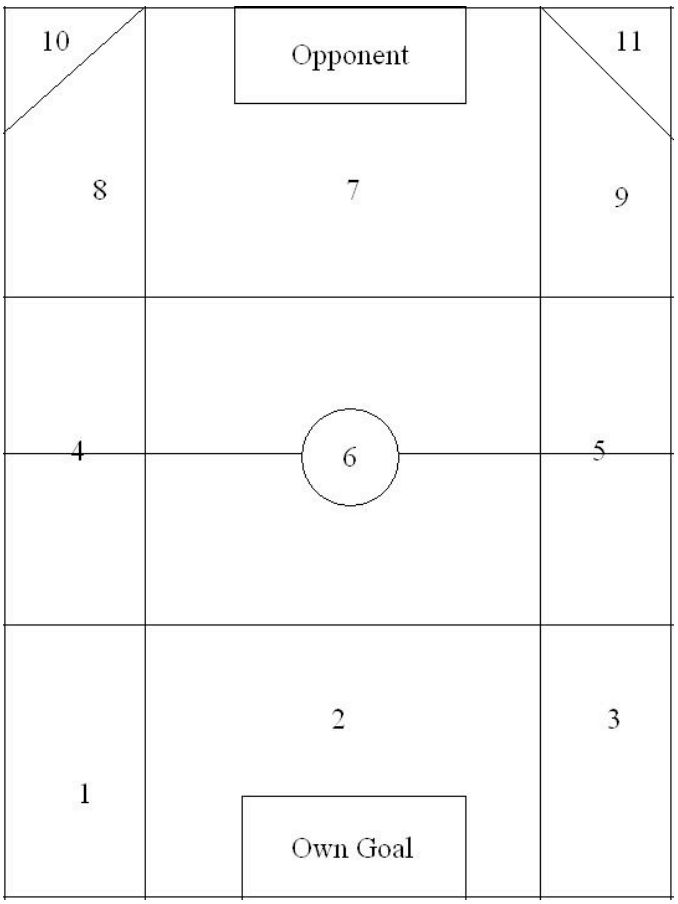

Figure 5.2: The field division. 1: Defensive left edge, 2: Defensive middle, 3: Defensive right edge, 4: MidField left edge, 5: MidField right edge, 6: MidField, 7: Offensive middle, 8: Offensive left edge, 9: Offensive right edge, 10: Offensive left corner, 11: Offensive right corner

5.4 Leaf Nodes

Leaf nodes store corresponding low level behaviour modules that we wish to execute. As we described early on these modules can be roughly classified into two groups: behaviors that needs require grab the ball, and behaviors that do not require ball grabbing.

The behaviours require ball grabbing are:

- Grab_Turn_Kick
- Grab_Turn_Shoot
- Grab_Dribble_Goal
- Grab_Dribble_Stop
- Grab_Upen_Left
- Grab_Upen_Right

Those behaviours that run without grab are:

- Dribble
- Upen_Kick_Left
- Upen_Kick_Right
- Avoid_Own_Goal
- Head_Kick_Left
- Head_Kick_Right

All these above behaviors are used in Robocup World Championship at Bremen, Germany. We will next describe when do we use what behaviour according to this year's specification.

5.5 Using Contested Ball Info

When ball is contested, we usually channel the tree into a separate branch. Contest ball is detected primarily by:

- Vision Obstacles/World Obstacle Map [22]
- IR Sensors [1]
- Front Legs Paw Sensors

[1]The used of IR and Obstacles will be discuessed in GrabDribble low-level behaviours in the later chapter.

We believe we are in a contested if all of the following is true:

- Any of three three detections return 1, AND
- The robot is close to the ball

And, we will continue be in ball contested state for the following 15 frames.

5.6 Decisions In Defensive Third

In our defensive area, we emphasize the priority of efficient ball clearing. In addition, we also want to minimize the chance where we accidentally run into the ball from the wrong side so that the ball gets knocked into our own goal. Moreover, if the ball is contested, we want to minimize the chance of the ball being grabbed by our opponent, hence we tend to use behaviors that clear the ball and do not require ball grabbing. Finally, we have taken out the get behind ideas in this area which was introduced by 2005 year. The reason is that get behind is terribly slow and is against our defensive philosophy. Practical experience against strong team shows they can easily grab the ball well and dodgy around our defenders before most of get behind is accomplished[2]. These lines of thoughts may be summarized as below:

For the actual implementation of behaviour decisions in defensive third, one can refer to sSelkick.selectKickInDefensiveThrid() in PyCode directory.

5.7 Decisions In Mid-Field Third

In mid-field our players start to attempt to grab the ball. We therefore encourage our player to grab the ball and compete with opponent when we are contested. Also, we have modified the decision tree such that it always attempt grabbing the ball if the robot is facing upwards. The structure of this tree is similar to the one we described with defensive area, but with more emphasize on the control of ball. We believe that being able to control the ball and pass it to the expected direction in the middle of the field is critical to our overall strategy.

The decision tree in this area when ball is contested only uses Upennkick. This is because we are confident about the reliability of this kick type, and

[2]For a complete get behind behaviour description, please refer to rUNSWift 2005: Road to Robocup 2005 by Nobuyuki[11]

Algorithm 5.2: Decision tree used in defensive third area by rUNSWift 2006.

Data:
Result:

```
1  selfH2CenterH = Angle from self position to (ballX,ballY+offset)
2  if If ball is contested then
3  |  return contestBallActions()
4  else if Ball on Edges then
5  |  return DecisionOnEdges
6  end
7  else
8  |  if 45 <= abs(selfH2CenterH) < 70 then
9  |  |  return UpenKick
10 |  if 10 < abs(selfH2CenterH) < 45 then
11 |  |  return HeadKick
12 |  if abs(selfH2CenterH) < 90 then
13 |  |  if abs(selfH2CenterH) < 20 then
14 |  |  |  return Dribble
15 |  |  else return GrabDribble
16 |  else
17 |  |  return GrabDribble
18 |  end
19 end
```

Algorithm 5.3: Contest ball decision tree by rUNSWift 2006.

Data:
Result:

```
1  selfH2CenterH = Angle from self position to (ballX,ballY+offset)
2  if 35 <= abs(selfH2CenterH) < 70 then
3  |  if selfH2CenterH < 0 then
4  |  |  return UPenKickRight
5  |  else
6  |  |  return UPenKickLeft
7  |  end
8  if we are about to grab or we are on the edge but facing roughly
   correct direction then
9  |  return GrabDribble
10 else
11 |  return Dribble
12 end
```

Algorithm 5.4: Ball near edges decision tree by rUNSWift 2006.

Data:
Result:
selfH2BallH = Angle between self and the ball, normalised to [0,360]
ballH2TGoalH = Angle between ball and target goal, normalised to [0,360]
if *20 <= selfH2BallH <= 160* **then**
  if *30 <= abs(selfH2CenterH - ballH2TGoalH) <= 80* **then**
    return UPenKicks
  else
    return HeadKicks
  end
else
  return GrabDribble
end

Algorithm 5.5: Decision tree used in field middle area by rUNSWift 2006.

Data:
Result:
selfH2CenterH = Angle from self position to (ballX,ballY+offset)
if *If ball is contested* **then**
  return contestMiddleField()
else if *Ball on Edges* **then**
  return edgesMiddleField()
end
else
  if *45 <= abs(selfH2CenterH) < 110* **then**
    return UpenKick
  if *20 < abs(selfH2CenterH) < 45* **then**
    return HeadKick
  else
    return GrabDribble
  end
end

Algorithm 5.6: Contest ball decision tree used in middle filed by rUNSWift 2006.

```
Data:
Result:
selfH2CenterH = Angle from self position to (ballX,ballY+offset)
if 35 <= abs(selfH2CenterH) < 70 then
    if selfH2CenterH < 0 then
        return UPenKickRight
    else
        return UPenKickLeft
    end
else
    return GrabDribble
end
```

moreover, robots rarely lose track of the ball after the kick. Hence Upenn kick is our primary kick in middle field when we are playing one on one, or even one on many opponent players in a contested situation. Its short motion duration, can help to quickly push the ball to places behind opponents up field.

Furthermore, we encourage our player to grab the ball and control the situation when the robots are facing roughly towards the attacking direction and of course when completely out of the direction. i.e. We need to grab the ball and adjust its rolling direction when we are chasing the ball towards our own goal. These lines of thoughts can be reflected by figure 5.6

5.8 Decisions In Offensive Third

Decision tree at the offensive third is much more complicated, for we need to consider various situations like contest ball, edges, corners. Moreover, we need to cater for on goal shot area close to opponent's goal, and special case like avoid grab the ball and dribble it across opponent's goal line. Offensive third is the region we tend to grab the ball in most of the situations. It is usually difficult to let the ball roll behind opponent robots and then still be able to chasing it up in a short amount of time. And within this region we tend to have a extremely contested situation, where opponent goalie, opponent defender, opponent attacker, our attacker and our supporter are settled in this small rectangular region. The most effective approach here, and which is used by teams that have strong ability of ball controlling in Robocup World Championship, is the idea of grab, dodgy, and shoot. In

Algorithm 5.7: Decision tree used in offensive field area by rUNSWift 2006.

Data:
Result:

selfH2CenterH = Angle from self position to (ballX,ballY+offset) selfH2BallH - Angle from self position to ball position ballH2CrossingH = Heading between ball position and the front of the targetGoal selfH2CrossingH = selfH2BallH - ballH2CrossingH

```
if If ball is contested then
    return contestOffensiveField()
else if Ball on Edges then
    return edgesOffensiveField()
end
else if Ball on Corners then
    if 30 <= abs(selfH2CrossingH) < 75 then
        return UPenKick
    else
        return GrabDribble
    end
end
else if Ball really close to the goal line then
    return Dribble
end
else if We are still very far from the target goal AND
abs(selfH2TGoalH) < 15 AND we are near the center width then
    return Dribble
end
else
    return GrabDribble
end
```

this area, teams with each other are not only competing about their decision on what low-level modules they need to run at strategically level, but also the goodness, efficiency and quality of their underline low-level skills such as grabbing, tracking, and kicking.

Decision tree for offensive, mid-field and offensive areas are all implemented in sSelKick.py in PyCode directory. The top level interface is called perform(), which in turn links to selectKickInDefensiveThird(), selectKickInMiddleThird() and selectKickInOffensiveThird() functions.

5.9 Summary

In this chapter we have outlined our approach on how we generate decision on what low-level behaviors to execute upon the current game situation. This approach is particularly implemented for our attacker player in a game, where we use a set of attributes to represent the game situation, and the decision system maps these attributes into a low-level behaviour to execute which is stored at the leaf nodes on the decision tree. This decision tree is fully implemented in sSelKick.py module inside PyCode directory.

Currently most of our input attributes only describe from the attacker's point of view. For example, the angle of the attacker to the target goal, the angle from the attacker to the ball. Future work might introduce more parameters that are related to other teammates, and even a rough idea on the opponent team players. For example, given the supporter's position, my current position, the ball's position and the opponent's players position, we can plan a path to kick the ball alone such that our supporter can run up and receive the passing.

Chapter 6

Low Level Behvaiours

6.1 Low Level Behaviour Overview

Low-level behaviour refers to behaviour modules designed to target specific task or tasks of similar nature. These modules sit at the very bottom of our behaviour system architecture and each written with a fixed format of API thus capable of running to play that task on its own. Similar behaviors are then grouped into single one module. These modules are called by high-level behaviors through decision making as outlined in the previous chapter. These modules then talk to actuator-control system that in turn controls the effectors and joints thus complete the behaviour specified in the physical environment. In this chapter, we will dig into the three most important low-level behaviors: seeing the ball, go and get the ball, and play while holding the ball. In rUNSWift terminology, these refer to ball finding and tracking, ball grabbing, and ball grab dribbling.

6.2 Find and Track Ball

Perhaps the most important skill in Robocup is find ball and track ball, that is, if I can see the ball, then adjust my head parameters to concentrate on it, is I can not see the ball, I need to quickly find it. Being able to see the ball is vital in a game. Teams that can consistently track the ball and being able to regain the ball upon lost have innumerous advantages in terms of speed, accuracy, and reliability. In this small section we will discuss our tracking and ball finding modules. Most of these modules were significantly re-written since year 2004, and were improved through years. Here we will discuss both the features introduced by the 2006-year team, as well as those important functional units that originated from previous years for completeness.

6.2.1 Ball Tracking

Ball tracking is the ability to point the robot's head to the ball, if one is seen on a vision frame. A good tracking implementation should minimize the occurrence of losing track of the ball, especially when ball is rolling and given the fact that there are other noises in the system such as reliability of visual ball detection from vision system and joint sensors fluctuation in the real game. Here we describe two approaches used at rUNSWift. Both approaches aim the top of the ball.

The first approach uses point projection that project the top of the ball on the image plain onto the image background. The robot then adjusts its pan, tilt, and crane to look at that projected point. This method is easy to implement and relatively reliable.

The second approach uses the distance from the ball and its relative heading from the robot's stance, calculates the required pan, tilt and crane to look at the ball.

Our comment on these two methods agrees with previous years. That is, the first method tends to perform better. The reason being is that the second approach directly rely on the ball distance and that distance is calculated by the ball radius from higher up vision ball fitting process. Consistently coming up with an accurate ball radius is somewhat challenging in practical. For example, part of the ball is not shown on the image corner or covered by obstacle on the vision frame. In practice, we use the first approach to direct our head onto the top of the ball and use the second approach only when our point projection fails to succeed.

Ball tracking function is implemented in hTrack.py module with interface calls: trackVisualBall(). trackVisualBall function will further call trackVisualBallByProj() which uses the point projection method, and trackVisualBallByTrig() which utilises the second method.

6.2.2 Ball Finding

Distributed Find Ball

If none of the robots on the team can see the ball, and it has been the case for a while, then the 3 forward players will perform a team play - distributed find ball. The strategy specifies positions for the forward players, and each player will go to its assigned point and perform spins on that point. The 3 points are chosen such that we can cover as much area on the field as possible. In this strategy goalie remains inside the goal box. The strategy

will cease if any of the robots including goalie re-capture the ball. The three points used in this year code are: (180,162), (80,378), (280,378).

The strategy also specifies which robot goes to which point. The robot closest to the bottom point (180,162) will go to that point. The robot that is closest to point (80,378) in the remaining two robots will go to that point. Finally the last robot will go to point (80,378).

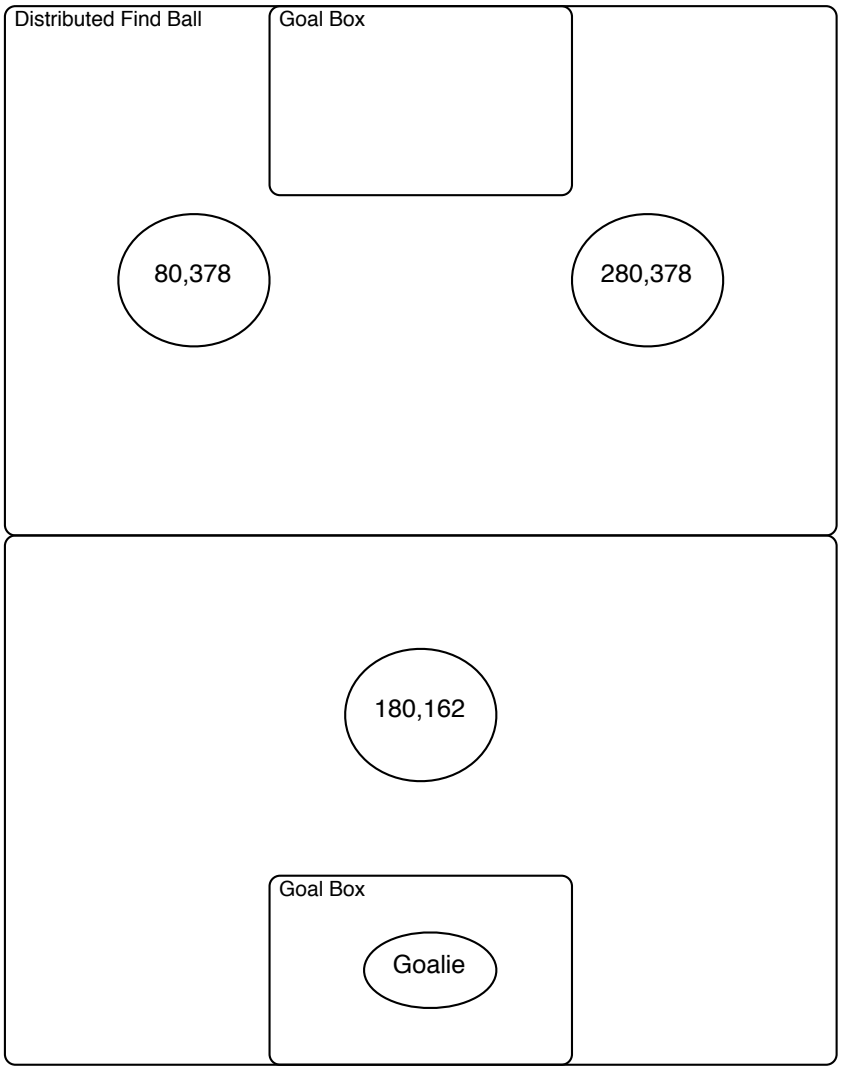

Figure 6.1: The 3 positions that the team will do distributed find ball on. The distributed find ball is triggered if none of the robot can see the ball for a specified duration, and is finished upon the a found ball event by any dog on the team including goalie.

General Ball Finding

The general find ball routine is triggered when an individual robot loses track of the ball. This is achieved through a sequence of actions. This sequence can be graphed as below:

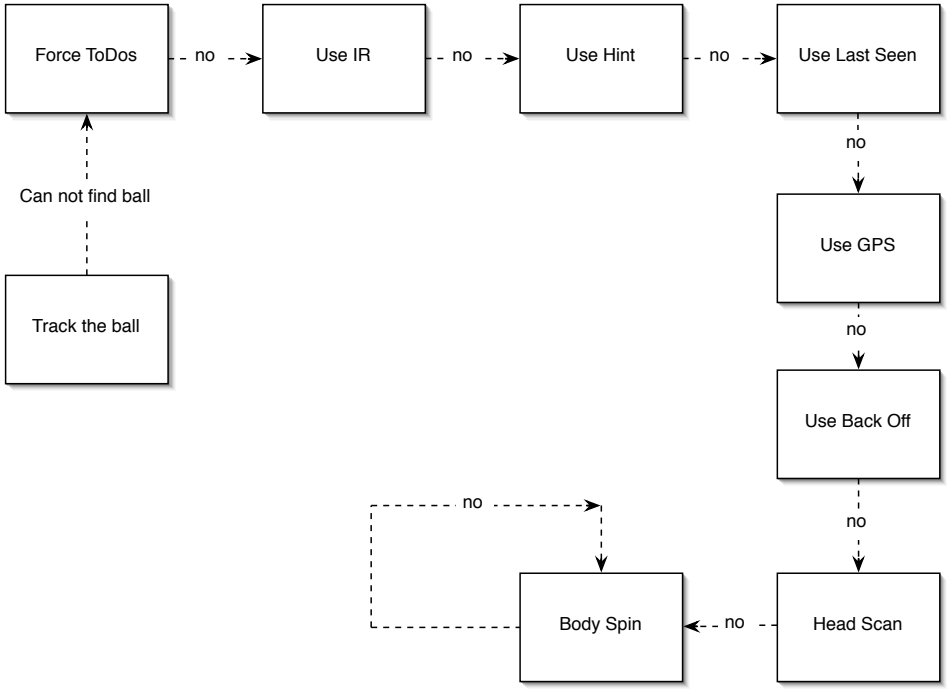

Figure 6.2: The sequence of find ball strategy we use if we can not see the ball. This sequence consist of useForce, useIR, useHint, useLastSeen, useGPS, useBackOff, useScan, useSpin.

Use Force

The first state is so-called "Force Todos", where we force the robots to perform certain find ball routines. Force state has a timer, which is used to determine within what amount of time after lost ball should execution of such routines be forced. Typical use of this semantic is to force robots look at the out direction of the ball after we just kicked for the duration of force timer. Another typical application is to allow the robot to walk to and look at its GPS ball position, which is the prediction of where the ball is.
[1]

[1]For a complete description of rUNSWift GPS system please refer to rUNSWift Undergraduate Thesis 2006 by Oleg Sushkov.

Use IR

In the next step we will check if there are objects very close in front of us. If so we need to have a quick look in that region. This check is carried out by means of check IR sensors. This check is necessary when the ball is direct under the robot's chin and is just off its vision. In IR state there is also a timer, which specify for how long we can remain performing this kind of search. The search for IR is triggered by:

- The IR timer has not expired
- The filtered IR value is high
- I am not blocking now

If IR is triggered, the robot will direct its head to the position with pan, tilt and crane parameters being (0, -10, -20) respectively. The robot will remain walking to where it think the GPS ball is.

Use Hint

Ball hint frame work was originally implemented by the 2005 team. The reason was originally to build a framework that is able to allow the robot to forget the normal find ball routine and jump to special find ball action sequence signaled by other behaviors. [11] A hint is represented by the distance and heading parameters with a time stamp. The time stamp is used to specify the valid duration since a hint is set by other actions. The distance and heading are used to specify the hint point in polar form. The find ball routine will first check if there are any hint set that is still valid. If so it will direct the head of the robot as well as its body to the hint point while the hint is still valid.

The hint framework is extensively used in our behaviour module. These modules include grab, grabDribble, headKick, and UpenKick.

Use Last Visual

The algorithm then moves on to check the time since the last visual ball. If the last seen ball was recent, then we will use the last visual ball's position to find the ball. This technique is extremely useful, because it allows the discontinuities of visual ball recognition. In practical find ball based on last visual ball is triggered when the last visual ball is within 4 vision frames

Behaviour Module	Hints Set (Dis,Ang)	Description
Grab	setHint(30,0)	We set this hint while we doing approach ball. If the grab failed then it is more likely we have knocked it away.
GrabDribble	setHint(35,-/+ 120)	If we are turning while we are carrying the ball, we set hint to the point such that the hint heading is opposite to our turning direction. This useful if we lost the ball while we spinning.
HeadKick	setHint(50,-/+ 30)	If we executed head kick, then we want the robot to look at the direction the ball going out to
UpenKick	setHint(25,-/+ 30)	If we executed Upen kick, then we want the robot to look at the direction the ball going out to

Table 6.1: The list of hints used in our behaviours

counting backwards from current frame.

If last visual ball was recent, then we will not move our head but leave it to where it is. That is, we will not move our head if we think the last visual ball is with past 4 frames. One exception occurs when we have just spun in the recent time. If such situation happens, the head's pan value is set slightly different from its old value. If the robot's spin direction was clockwise, then we set the pan to be its old value + 30 degree in anti-clockwise direction. Otherwise we set the pan to be its old value - 30 degree in the clockwise direction.

The use of last visual ball is a mean of putting hysterisis on our head motions. Here we have allowed discontinuity of 4 noise vision frames to decide if we have really lost track of the ball or not.

Use Gps Ball

When last visual ball was not very recent, the robot moves onto a stage where it starts to find ball relying upon GPS ball information. Again the constraint used here is that the time since the last time we see the ball is shorter than 26 vision frames.

Use Backing Off

Using backing off refers to the behaviour where the robots step backwards first, then proceed find ball from there. The reason is to allow perceive ball very close to the robots. For example, directly under its chain. The trigger of this stage is that the distance from the robot to the mean of GPS ball is shorter than 40 cms.

When performing backing off, the robot will step backwards with maximum speed, and perform the standard head scan defined in hTrack.py

Use Head Scan

If none of the above tricks can be performed, the algorithm moves on to make the robot scan its head looking for the ball. The scan used here is a slight modification of our traditional full scan. Where we limited the full length of the pan to the size of the variance of the ball. The idea is, if the GPS ball variance is small and the variance of the robot's position is also small, then we are relatively confident to scan in narrower range towards to that direction. On the other hand, we would scan a broader range by using a corresponding larger pan range. The robot should remain scanning for the ball until it exceeds the scan state timer. This scan is therefore called variance scan. The time limit used here is 30 frames. That is, robot will remain in this state for entire duration of 1 second unless it finds the ball.

Variance scan is implemented on top of our traditional full scan, but with specified minPan value and maxPan value according to the ball variance and robot's variance. The interface of our full scan is given by:

*s*can(lowCrane,highCrane,lowTilt,highTilt,lowSpeed,highSpeed, minPan,maxPan,lookDown=False)

The calculation of the minPan and maxPan is given by algorithm 6.1:

The full scan call resulted from this calculation is then given by:

scan(lowCrane=-10,highCrane=10,lowTilt=0,highTilt=0,lowSpeed=8, highSpeed=9,minPan=ballH-theta,maxPan=ballH+theta, lookDown=False)

Use Spins

Eventually, the robot will go into spin state if none of the above tricks help to find the ball. A spin means fixed head motion with body spin only. The

Algorithm 6.1: The calculation of minimul and maximum pan value according to the variance of the robot and it s GPS ball.

Input: scale = 1.0
Output:
Result: minPan, maxPan

```
1 total_var = sqrt(abs(GpsSelfMaxVar)) + sqrt(abs(gpsBallMaxVar))
2 theta = atan(total_var, ballD) * scale
3 if ballH > MAX_PAN - theta then
4 |   ballH = MAX_PAN - theta
5 end
6 if ballH < MIN_PAN + theta then
7 |   ballH = MIN_PAN + theta
8 end
9 return (ballH-theta,ballH+theta)
```

head parameters are locked for 2 situations.

- GPS ball distance is grater than 120 cm
- GPS ball distance is less than or equal to 120 cm

The typical setting we used are:

- Action.setHeadParams(45 * spinDirection, -10, 0)
- Action.setHeadParams(30 * spinDirection, -10, -10)

The angular velocity is also set to be variable, with respect to the distance of the GPS ball. This algorithm is presented in 6.2:

As we can see the returned angular speed for spinning depends on the GPS ball distance. If the ball distance is far away, for example, 300 cm, the angular speed returned is 55.0. If the ball distance is close, say 20 cm. The angular speed to be used would be 60.5, with increased magnitude. The intuitive behind this is that the team feels more confident about picking up visual ball in the closed and medium range. As the ball rolls away from the robot, such confidence depredates, especially in a complicated game environment. We therefore specify the robot to spin slower when the ball is far away.

Algorithm 6.2: Variable angular velocity used by find ball spinning

Input:
Output:
Result: Spin angular speed

```
1  MAX_BALLD = 120.0
2  MIN_BALLD = 20.0
3  Spin_Speed = 55.0
4  i = float(MAX_BALLD - Global.ballD)/float(MAX_BALLD -
   MIN_BALLD)
5  if i > 1.0 then
6  |  i = 1.0
7  end
8  if i < 0.0 then
9  |  i = 0.0
10 end
11 MIN_DIST_SCALE = 1.0
12 MAX_DIST_SCALE = 0.9
13 dist_scale = MIN_DIST_SCALE + i*(MAX_DIST_SCALE -
   MIN_DIST_SCALE)
14 return Spin_Speed * dist_scale * spinDirection
```

6.3 Grab

6.3.1 Introduction

In this section we will discuss another very important skill, ball grabbing. Ball grabbing refers to the behaviour the robot holds the ball using its two front legs and its head. Ball grabbing is especially beneficial in controlling the ball and handle complex motions while carrying the ball. Kick executed on a grabbed ball tends to be much more reliable then those kick executed on the fly. In addition, ball grabbing also makes grab dribbling feasible, that is, carries the ball while running. This is later so-called grab dodge and shoot strategy used extensively by those top teams in this year's Robocup competition.

The earliest record of grab behaviour goes back to 2003/2004. Where the field wall still exist, so being able to accurately grab the ball is not an extremely important factor in a game. In year 2005, field wall is taken off from the filed, so it is possible to kick the ball out of the field area. Since then, teams started to see the importance of this ball controlling behaviour. People have been working on optimizing their grabbing behaviour, some team used the grab to stabilize their kick, and some other teams used this technique to carrying and shoot the ball. Interestingly, it turns out that high

quality performance of grabbing behaviour requires quality work on nearly most of the other modules. A good ball distance is required from the vision, correct ball velocity and global position are needed from localization, and finally approaching ball precisely requires top-quality work on the underline odometry walking engine.

To target the reliability of grabbing behaviour, many teams have sacrificed their performance from other motions. Some team will wait to grab until the ball slows down, some other teams would slow down their approaching ball motions to get balance between the grabbing accuracy and some hidden defects in their odemetry calibration. rUNSWift 2005 grab was successfully developed and extensively used for its moderate reliability. However, there are some known factors need to be reviewed this year. First of all, slowing down by significant magnitute when approaching to the ball is not desirable. Secondly, the side adjustment steps also delayed our ball approach time. The lost in the semi-final with our friend team NuBots in Robocup 2005 is a good example of how much difference can this compensations make.

In year 2006, our directive comes similar with NuBots, that is, avoid slow motions while maintain the reliability when approach the ball, and this forces the team to look at works on optimizing modules in vision, GPS, as well as underline walking engine. It turns out that those optimizations were all complete with high quality and these resulted a great improvement in our grab by a large amount as can be seen in the Robocup 2006 Championship. Compare with 2005. This year's grab motion is much faster, and more reliable. This ability enabled us to compete confidently with those top teams in a complex game environment.

6.3.2 Module Layout

The grab module consists of three logical functional units. Approaching, Grabbing, and a query function of "Can I grab now". Approaching function gathers the information out the ball, in particular the distance and velocity, and returns the forward step amount, left step amount and turning amount to step to the underline walk engine for the current frame. Grabbing function executes a fixed sequence of actions in time space. It starts by pointing robot's head forward out, and ends up with dropping its head on the ball over a specified amount of time duration. The query function returns true if it is appropriate to start grab by calling grabbing function at this frame.

Tuning each of these parts would affect the final performance of the whole grab module as a whole, and change in other modules can also result

in performance degradation in the grab. This year the team has spent enormous amount of time to keep maintain this module.

6.3.3 Ball Approaching

The ball approaching function's main role is to calculate at each frame the forward, left and turn component for our underline walk engine. This year we removed the side step line up mechanism used in 2005 for speed reason, that is, we decide to use zero left component in grab approaching. Instead, the robot will always be walking towards the heading of the ball, and this is done through a sequence of specification of only the forward and turn components. The simplest form of this function that could be:

$$\text{walk(BallDistance} * \cos(BallHeading), \text{ 0.0, BallHeading * 0.9)}$$

The magnitude forward component is limited by our turning amount. If lots of turn are needed, the forward component given to walk engine is limited by the order of Cosin(BallHeading). On the other hand, if the robot is facing directly to the ball so the turn component is close to zero, we will use the full forward strength instead. In addition, we have reduced the ball heading by 10% to prevent over turning effect.

6.3.4 Can Do Grab Query

As the robot adjusts its forward and turn to line up and chase the ball, the robot is also checking to see if it is appropriate to execute grab ball action at the current frame. The implementation of this check is to estimate if the ball will intercept with the robot in the next few frame by using the ball velocity information.

We say the ball is intercept with the robot and hence is appropriate for the robot to start grabbing when:

- The horizontal displacement between the ball and robot is less than 3 cm, AND
- If we have done a large amount of turn in the last 15 frames while approaching, then the ball distance must be less than 9 cm, AND
- Otherwise the ball distance must be less than 13.5 cm.

The entire algorithm is outlined here:

Algorithm 6.3: Checks if it is appropriate to start grab

```
   Input:
   Output:
   Result: Yes/No
 1 ballX,ballY = ballPosition()
 2 ballVX,ballVY = ballVelocity()
 3 PredictionItr = 6
 4 itr = 0
 5 GrabDistance = 13.5
 6 while itr < PredictionItr do
 7 |   ballDistance = sqrt(ballX * ballX + ballY * ballY)
 8 |   can = true
 9 |   can = can AND abs(ballX) < 3
10 |   if FrameSinceLastSpin > 15 then
11 |   |   can = can AND ballDistance < GrabDistance
12 |   end
13 |   else
14 |   |   can = can AND ballDistance < GrabDistance * 0.75
15 |   end
16 |   ballX += ballVX
17 |   ballY += ballVY
18 |   ballVX *= 0.90
19 |   ballVY *= 0.90
20 |   itr++
21 |   if can then
22 |   |   return true
23 |   end
24 end
25 return false
```

Notice that the degradation of ballX and ballY velocity at end of each iteration is used to model roughly the friction effect.

6.3.5 Grabbing

The 2006 grabbing motion completes in 12 vision frames, which is more than 2 times faster than 2005 grab motion. (25 vision frames) The function will drive the head and the legs over these 12 frames to complete the grab. This leaves us to about 2.5 seconds to perform dribbling and kicking after the grab. In fact, for some situation the grab can be completed in an even

shorter time duration if infrared sensor distance check is triggered. This could in theory double our grab speed, since we increment the counter twice as fast when infrared sensor check is triggered. Using infrared sensor in grab allow us to grab close in ball reliably.

The grab motion starts by lifting the head of robot up a slightly, allowing the ball to go in. The robot will also walking towards to the ball and drop its head at the end of the duration as well as open its mouth.

The entire 12 frames are divided in the following ways:

State	Duration	Description
Grab Forward	0 - 3 frame	Lift head up and walking with maximum forward speed without any left and turn compoents.
Grab No Check	3 - 6 frame	Drop head on to the ball but keep mouth closed
Grab Complete	6 - 12 frame	Keep head motion as above, open mouth

The entire algorithm of grabbing motion is given by figure 6.4:

Summary

Grabbing can be considered as a form of the "chicken and egg dilemma", in the respect that to play a "grab and move" style requires good grabbing, but good grabbing is only important in a "grab and move" style. [21] Since year 2004, rUNSWift has adopted the "grab, dodge and shoot" style of game play, and the team has been striving for the accuracy, reliability and speed since then. Over the years, the grab reaction time becomes shorter and shorter, the quality of the grab becomes higher and higher. Inspirited by NUbots, the 2006 team removed side walk line up and re-wrote the grabbing motion to accelerate the grabbing speed. At the same time, the team has also paid special attention on maintain the reliability of the grab module. This fast and reliable grab has gained the team great advantages in the Robocup 2006 World Championship.

Algorithm 6.4: The grabbing motion in grab forward, grab no check and grab complete time durations

Input:
Output:
Result:

```
1  walk(MAX_Forward, 0.0, 0.0)
2  grabbingCount = frameSinceFirstGrab()
3  if IRSensor > SensorNoiceLimit then
4  |  grabbingCount += 1
5  end
6  if can see ball then
7  |  lastSeenBallHeading = BallHeading
8  end
9  if grabbingCount < 3 then
10 |  setHeadPara(pan = lastSeenBallHeading, tilt = -60, crane = 70)
11 end
12 else if grabbingCount < 6 then
13 |  setHeadPara(pan = lastSeenBallHeading, tilt = -40, crane = 50)
14 end
15 else if grabbingCount < 12 then
16 |  setHeadPara(pan = lastSeenBallHeading, tilt = -40, crane = 50)
17 |  openMouth()
18 end
19 else
20 |  return Completed
21 end
22 return Executing
```

6.4 GrabDribble

6.4.1 Introduction

GrabDribble refers to the behaviour of carrying the ball around and shoot on goal in case of appropriate. The total time duration allowed to grab and grab dribble is capped at 3 second and the robot is not allowed to move by more than 50 cm in displacement. GrabDribble is the most important and complicated high-level behaviour module at rUNSWift. This module provides complete action specifications after grabbing is succeeded. These include, carrying ball forward, turning and aiming, obstacle avoidance, kick and grab-dribble kick selection tree.

Grab-Dribble module at rUNSWift was originally written by Nubuyuki in 2005. The 2006 year team used very similar structure but has also proposed many useful optimizations. Large amount of work was conducted in aiming, obstacle avoidance and kick selection tree.

The team has decided to abandon GPS as the primary aiming method in defensive half. Aiming is now nearly all done by using visual information. The robot would only need to aim GPS target in case of failing to recognize it visually. Empirically this increased the accuracy of our aiming significantly. The robot is able to consistently perform the action sequence of grab, turn, aim and shoot.

In addition, the team has re-written dodging behaviors, this include introduction of using infrared distance sensor as primary obstacle detector in combination with visual obstacle as secondary. Moreover, separate specification of strafing goalie and strafing other opponent players were used. This turns out to help to score most of the on-goal shots when against teams that have strong goalies.

Furthermore, the team has modified the primary kick used in 2005 from headTap kick to fastForward kick. This modification retained the characteristic of strong strength of this kick type but reduced the reaction time by great amount. The combination of strong long-range on-goal kicks with several other short-reaction soft kicks made the team's attack more smooth and efficient. A special kick type, "head kick" was also added to the kick selection and helped scored against GermanTeam in a practical game.

In the following paragraphs we will outline the working procedures of this year's GrabDribble module, discuss its pros and cons, with emphasize on the above optimizations.

6.4.2 Turn and Aim

Before handling those fancy features such as dodging and shoot on goal, we first of all need to reliably line up with the goal, in particular, we need to reliably line up with the gap between goalie and one of the goal post. These insights naturally lead us to answer the following questions. Where do we aim, Which way do we turn, and When do we stop.

Use Visual Target Aiming

Visual target aiming is much more reliable than GPS aiming. In visual aiming, the relative heading between robot and target is provided by underline vision module, and the robot simply need to turn that amount to face the target. This year we encouraged visual aiming and discouraged, actually took out as much as we can about GPS target aiming. Our confidence comes from the reliability and advantages of many other low level modules. In particular, the speed and robustness of vision module and advantages of our grab-dribble stance. Robot can reliably see the target goal from over half of the field on grab-dribble stance. This allowed us the ability to line up with goal from far distance. The visual goal recognition is completed in frame, hence our robots can adjust its heading on a frame by frame basis.

Aiming Procudures

If robot can not see the target goal, it needs to turn to look for it. If it can see the goal after a few frames then it will switch to aim one of the gaps rather then the center of the goal. Robot will keep spinning until it sees the goal when it is at the the front half of the field. The reason here is that the chance of spin in front of the goal within that range and not seeing the goal within any frames when robot is heading to it is exetremely low. In a game, and we have tested over and over again, the idea of keep spinning until sees the goal is highly successeful, especially when GPS heading fails to return sensible value.

The decision on which way to turn for goal searching is determained by the robot's GPS information. We would like to turn in the direction such that the angle starts from the robot across that direction to the target goal is smaller than going opposite way. While turning a constant angular velocity is used. In practice we used walk(0, 0, 50.0) when robot has its Y position component less than 400cm, and used walk(0, 0, 40.0) otherwise.

As we mentioned above, if visual goal is detected on the current frame, we would prefer to aim the gap but not necessary the goal center. To-

gether with the heading of the goal, we also calculate the two gaps seen on the goal, this two gaps are represented by the ((leftHeading, rightHeading), (leftHeading, rightHeading)). All are relative headings from robot point of view and abs(leftHeading - rightHeading) represents the gap size. The aiming algorithm at this point will pick up a gap and lock it. This gap selection algorithm selects the gap based on its size and is given by:

Algorithm 6.5: The gap selection logic used in 2006, if we have already locked one gap then return that one. If we have not selected one gap then select the wider gap from the two gaps.

Input:
Output:
Result: bestLeftHeading, bestRightHeading

```
1  global locked_gap
2  if NOT locked_gap = NULL then
3  |  return locked_gap
4  end
5  (lmin, lmax, rmin, rmax) = getHeadingToBestGap()
6  if lmin=0 AND lmax=0 then
7  |  return (rmin,rmax)
8  end
9  else if rmin=0 AND rmax=0 then
10 |  return (lmin,lmax)
11 end
12 else if abs(lmin-lmax)>abs(rmin-rmax) then
13 |  return (lmin,lmax)
14 end
15 else if abs(rmin-rmax)>=abs(lmin-lmax) then
16 |  return (rmin,rmax)
17 end
```

Once the desired gap is locked, the robot will direct its direction to the middle of the gap. To determain if robot is in fact aimed successfully with the gap, we calculate the average heading of the gap and come up with a padding proportional with the size of the gap. We then say the robot is aimed at this frame if:

$$\text{abs(average\_gap\_heading)} < \text{padding}$$

The entire gap aiming algorithm is given by:

Algorithm 6.6: Gap aiming creteria, where if accumulated aimed frame reaches 4 then we say the robot is aimed

Input: BestLeftGapHeading, BestRightGapHeading
Output:
Result: Complete of Gap Aiming

```
1  global aimCount
2  middle_of_gap = abs(BestLeftGapHeading + BestRightGapHeading)
   * 0.50
3  padding = abs(BestLeftGapHeading - BestRightGapHeading) * 0.25
4  if abs(middle_of_gap ¡ padding) then
5  |  aimCount ++
6  end
7  if aimCount > 3 then
8  |  return Succeeded
9  end
10 return Executing
```

6.4.3 Apply OverTurning

When attempting to line up with target goal at defensive third back yard, i.e. Y less than 200 cm, we need to start using localization to direct our turn component. This is because we are relatively less confident in this range for robot to spin and not miss seeing the goal. The logic is modified slightly to accommodate the situations. Our robots will still turn to the same direction as we outlined above. If it captures the target goal then we switch into accurate gap aiming. If our robot has not seen the target goal but GPS localization believes robot is actually facing it, we allow our robot to over turn further for a few frames before stop. Empirically we have seen too many times where the robots turn and attempt to line up purely on its GPS localization and stopped just before it could see the target goal. There are situations where the robot overturns and also missed all visual target from long distance. But compare with fixing much more under turn situations, this is relatively minor especially when we require less accuracy such as lining up from far away range and not executing for on-goal shoot after that.

6.4.4 Obstacle Avoidance - Dodging

Obstacles refer to objects on the field that hinders the progress of the attack. Obstacle avoidance is the ability to avoid any obstacles that could potentially obstructs the movement of the attacking robot or its shot. This could be an opponent robot, a beacon, or even a teammate player. This skill consists of obstacle detection and obstacle dodging.

Obstacle Detection

Obstacle detection is done primarily by vision and infrared sensor. Visual obstacles are recognized by their low Y value in YUV color space. If a pixel on a scanline falls below 35 then it is classified as obstacles. This approach was implemented by Alex North in 2005 and differs from most other attempts. [4],[20], [21] The total number of obstacle inside a 50cm by 100cm rectangle in front of the robot is counted and the obstacle avoidance will be triggered if this number is above certain threshold.

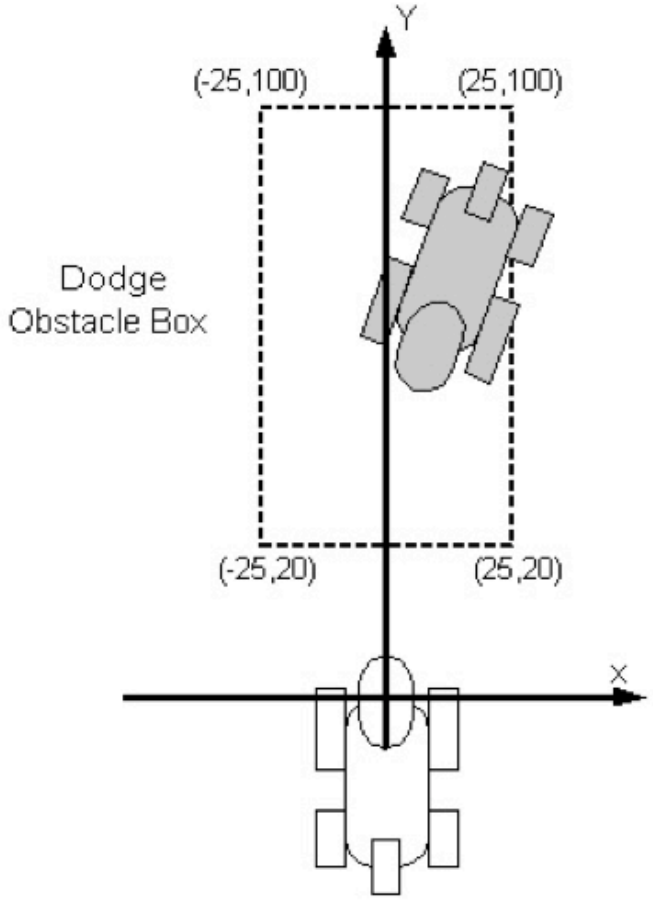

Figure 6.3: The traditional visual obstacle detection by counting discrete obstacle feature pixels in the rectangle projected on the field. Figure taken from Road to Robocup 2005.

The second method is to use head infrared sensor. At 2006 Robocup

most of the close in obstacle avoidance is triggered by this method. This method is very reliable to tell if there is obstacle in front of you in a yes or no format, irrespective of lighting condition. The sensor also helps to detect goal post, referee legs and beacons. [21]

The value returned by this head infrared sensor is a distance indication. The closer the obstacles are to the robots, the smaller this value is. To come up with a good obstacle threshold, one often need to come up with the "avoid-ground-value", that is, the value when the distance across to the ground in average. The threshold is then set to:

$$\text{IR Dodge Theshold} = 0.8 * \text{avoid-ground-value}$$

Interestingly, it turns out that the reading of IR value for fixed distance differs with the field surface material. i.e. for two pieces of field blanket, one can expect different such threshold. Moreover, it can be shown that different robots have different "avoid-ground-value", this could be possibly due to the physical differences among the sensors, as well as differences in physical joint angles reflections. For example, Nubots claims that their IR sensor values could range from 20cms to 25cms. [21]

To calibrate infrared sensor for obstacle avoidance, one can manually come up with different IR threshold for each Mac address and use each one for the corresponding dog. This requires separate tuning for every robot on a team. Another approach is to select a robot as the base robot. The rest of the robots' IR value will be modified by its separate correction model so that the final reading of these robots are same as the base dog per distance per environment. Now to calibrate IR threshold on another field, one can only estimate the new value for this base robot. In practice we used the second approach, where the correction model was a linear adjustment with X coefficient and a constant for each robot, and the IR dodging threshold used in the Robocup 2006 Ground Final was 20cm.

Obstacle Avoidance

Once the dodgy behaviour is triggered, robot will start side steps on a selected direction, for a calculated duration. The decision on which way to dodge is hard. In practice we have separated situation of executing dodge on opponent goalie and cases else wise.

The decision for dodge regular obstacles can be summarized as: If robot is near left edge, including top left corner, we always dodge to the right. If robot is near right edge, including top right corner, we always make it dodge

to the left. Moreover, in defensive half we always dodge to the near side. In offensive half we always dodge to the center of the goal.

Algorithm 6.7: Decision on dodgying direction

```
   Input:
   Output:
   Result: Left,Right
 1 if selfX<60 OR At TopLeftCorner then
 2 |  left = -LEFT
 3 end
 4 else if selfX > FIELD_WIDTH-60 OR At TopRightCorner then
 5 |  left = LEFT
 6 end
 7 else if selfY > FIELD_LENGTH * 0.5 then
 8 |  if selfX > FIELD_WIDTH * 0.5 then
 9 |  |  left = LEFT
10 |  end
11 |  else
12 |  |  left = -LEFT
13 |  end
14 end
15 else if selfY <= FIELD_LENGTH * 0.5 then
16 |  if selfX > FIELD_WIDTH * 0.5 then
17 |  |  left = -LEFT
18 |  end
19 |  else
20 |  |  left = LEFT
21 |  end
22 end
23 return left
```

The decision for dodge opponent goalie is slightly different. The difference comes from the fact we have the gap information. Here, an extension is added such that if the right gap size is bigger than the left gap size, then dodge towards to the right gap. If the reverse holds, we will direct the robot to dodge to that left gap. If no gaps are recognized, the robot will always dodge to the center of the target goal, increasing the likelihood of scoring.

The robot will dodge with the selected direction until the obstacle value drops back and maintains the status for a few frames. In practice robot will stop dodge if accumulated frames for which the obstacle features are below the triggering threshold has reached DODGE_RELEASE. In practice

we used 7 frames for goalie dodging and about 30 frames for dodge other players on the field. For example, to dodge goalie, every frame with obstacle lower than triggering thrshold, the robot MUST dodge for the next 7 frames.

6.4.5 Edge Behaviors

When handling grab-dribble near the field edge, a special function designed for this will take the control over. This function directs robot to move away from the field line using only backward walk, side step walk and diagonal walk. This behaviour is helpful in preventing carrying the ball over the field edge when turning. This function was implemented by Nobuyuki in year 2005, and is included here for completeness.

6.4.6 Kicks

In this sub-section we will discuss the four primary kicks used at rUNSWift 2006. These four kicks are hardForward Kick, Wedge Kick, HeadSwipe, and UPenn Kick. hardForward and Wedge Kick require ball in possession first. HeadSwipe and Upenn Kick can be executed with or without grabbing the ball first. Also, hardForward and Wedge Kick kicks in straight line and the other two types do not.

HardForward Kick

HardForward is the strongest kick we used in Robocup, though it is not the strongest kick we have in our code repository. The reason is that this kick was developed to target motion speed, and reliability. We view these two factors more important than strength. Fast kick motion speed minimizes the chance for opponents' push and disturb. The shot that has strong power and wide batting area would have little sphere of activity, if the shot was long motion. [35] Nowadays all on-goal shots are completed with external pressures.

In parallel with optimizing speed, we have also paid special attention on the reliability of hardForward kick. Reliability refers to the successful rate of the kick type. Moreover, the successful rate when opponents are pushing. Even loosely speaking, how often the ball really goes forward after the full execution of the kick, especially under external stimulis. We see shots in similar range used by top teams generally have all these characteristics. They just never fail the kick - Every kick would almost mean a forward displacement of the ball.

The hardForward kick has strength about 2.5 meters. We believe this is usually enough for a fast and reliable on-goal shot, as well as a quick ball clear used in defensive area.

Wedge Kick

Wedge Kick was developed by Andy Owen and was originally designed as a kick used in practical games to protect damage to the robot's head made by using those head kicks. This kick is a straight-line kick but with short targeting range. It goes to about one meter. Strategically this kick is equivalent to the NUbots's soft kick, and is used if the robot wants to release the ball in a very small distance.

This kick is used as our primary choice of short-range kick for its robustness. Its kick execution rarely fails. WedgeKick takes 12 motion frames to finish. However we believe this is still slower than its soft kick sibling in NUbots, which could finish in under 10 frames.

HeadSwipe Kick

HeadSwipe was developed by Oleg Sushkov. HeadSwipe is able to kick the ball hardly "on the move", that is, this is not a grab needed kick. The idea is to approach the ball and swipe the head of robot and knock onto the ball with the head. This kick is not as reliable as WedgeKick but has the advantage for its speed and efficiency. It is used more often in the mid-field to quickly clear the ball to the forward in a contested situation.

UPenn Kick

UPenn Kick [36] is another form of non-linear kick used in conjunction with HeadSwipe. Upenn Kick is reliable but slower than HeadSwipe.

6.4.7 Kick Selection Tree

At the end of the grab-dribble behaviour, the robot would select a kick to execute. This kick is selected based on this grab-dribble selection tree function, which outputs the most appropriate kick type to execute. This selection tree has three basic braches, defensive, mid-field and offensive.

Defensive Kick Selection

This branch deals with kick selections in defensive third of the field. This can be summarized as: "Avoid scoring own goal, and clear the ball quickly as we can". The kick we used is HardForward kick, so it is strong enough to effectively clear the ball. Also, if we could see our own goal, no kick will be executed, rather, robot will be forced to spin away from our own goal.

Algorithm 6.8: Kick selection in defensive third. We used simple strong version of linear kick conjuncted with own goal avoidance.

Input:
Output:
if *Own Goal Visible* **then**
  Spin()
end
else
  HardForwardKick()
end
return

Mid-Field Kick Selection

The kick selection tree in this region is similar in concept with defensive half, that is, avoid own goal and push the ball up field with strong kick. But we have also taken into consideration of ball-contested situation. If contested, usually a strong kick is less effective than a soft kick. We therefore added in the use of WedgeKick if ball is contested.

Offensive Kick Selection

The selection tree used here is a bit aggressive. That is, even when we are not seeing the target goal, we still want to shoot in some of these cases. At the first glance this might not make much sense, but experimentally at the exact frame we are about to shoot on goal, robot might or might not recognize the target. The trick we play is to use stronger kick when we can see the target, and as our confidence drops we decrease the strength of kick we select so to minimize kicking the ball off the field or kicking on the wrong direction. The kick selection tree used for offensive is therefore channeled in to three braches: Goal seen, Goal recently seen, Goal not seen.

The case of we can see the goal, we need to check if robot's head is within the gap. Moreover, we also want to release the ball immediately if

the robot is too close to the target goal.[2] In addition, we have channeled this tree down to another two special branches. These special cases consider the offensive kick at two top corners (100cm radius from the top left field point and top right field point), if the robot is in one of these corners and can see very small size of gap, it will give up the aimed kick and pass the ball cross the target goal instead. Putting all of these together, the offensive decision tree when target goal is visible is given by:

Algorithm 6.9: Kick selection in offensive with target goal visible.

Input:
Output:
if *On corner area AND Gap is small* **then**
  Pass Across()
end
else if *Still aimmed in gap* **then**
  HardForward()
end
else if *Goal Heading is within 15 defree* **then**
  HardForward()
end
else if *Distance to target goal < 30cm* **then**
  Release()
end
else if *Robot's Y position component > Field_Length - 50* **then**
  HardForward()
end
else
  WedgeKick()
end
return

The recentness of goal seen is measured by the frame difference of current frame and last seen target goal frame. In practice if this difference is less than 15 we say the robot has recently seen the target goal. In this case use strong kick when close to the goal, and use soft kick in all other cases.

In the last branch, the robot has not seen the target goal in the last 15 frames. If this is the case, then it is likely that the robot has not lined up with the target goal yet and still on its way of spinning. Empirically speaking, the robot has spun across the target goal and happened to missed all the visual target is very small. The kick command is still issued usually

[2]The 2006 soccer rule disallows carrying the ball over the line.

Algorithm 6.10: Kick selection in offensive with target goal recently seen.

Input:
Output:
if *On corner area AND Gap is small* **then**
  Pass Across()
end
else if *Robot's Y position component > Field_Length - 50* **then**
  HardForward()
end
else
  WedgeKick()
end
return

because robot is out of grab-dribble time limit. (3 second) Interestingly, this scenario is somewhat common in Robocup. Image robot A has the ball grabbed and is spinning to aim the target goal. Opponent robot B happens to come alone and pushes robot A. The result is that A can hardly spin because of the jam, or effectively, can only spinning slowly. In this case it is possible that robot A runs out of time before it could see the target goal. The second case we often see in a game is the robot selects the wrong direction to spin, due to the errors in localization. This could also make it run out of time before it could have chance to see the goal and line up with it.

The trick we play here is to turn on our HeadSwipe kick. HeadSwipe is a strong kick that kicks the ball out about 30-50 degree out from the robot's origin. Robot would HeadSwipe the ball to the right, if it is at the font third and has been spinning in the clock wised direction and still hasnt seen the goal yet. Robot would HeadSwipe the ball to its left, if it is at the font third and has been spinning in the anti-clock wised direction and still hasnt seen the goal yet.

Finally, robot will perform soft kick if it is at the front half of the field, and release the ball immediately if it realizes it is in defensive half field. The entire selection tree in this branch is given by figure 6.11:

6.5 Summary

The low level behaviors are certainly the most critical elements in Robocup. In a grab-dodge-shoot fashion, robot needs to be able to track the ball reli-

Algorithm 6.11: Kick selection in offensive with target goal not seen.

```
   Input:
   Output:
 1 X,Y = Robot's Postion
 2 if Y > 350 AND OwnGoal not visible AND
 3 for the past 30 frames, robot has been spinning then
 4 |  if TurnDirection = Clockwise then
 5 |  |  HeadSwipe(right)
 6 |  end
 7 |  else
 8 |  |  HeadSwipeleft
 9 |  end
10 end
11 else
12 |  if Y > Field_Length * 0.5 then
13 |  |  WedgeKick()
14 |  end
15 |  else
16 |  |  Release()
17 |  end
18 end
19 return
```

ably and re-capture the ball as quickly as possible if lost. We see ball finding and tracking being the foundation in the overall low-level behaviour. Next, we emphasize the speed, reliability and accuracy of our grab motion, evaluating if we can grab the ball reliably with as small amount artificial slowing down as possible, under complex game environment. Been able to track the ball and grab the ball well together gained the team enormous benefits over other teams and is certainly the step stone for handling post-grab motions, grab-dribble. In grab-dribble, we evaluate the surround environment to decide if robot should shoot on goal or dribbling to a even better position to shoot on-goal. In areas other than opponent target goal, we also use this dodging skill to avoid obstacles on the field and push the ball up field through the cleared region. Finally, we perform the combination of long and short range kicks, emphasizes the balance between strength and accuracy, speed and reliability.

The quality of these low-level behaviour modules heavily rely on the rest of modules in the system, in particular vision, localization and actuator controls. High standards originated from these modules made many of these nice features feasible in writing our behaviors.

Chapter 7

Behavious Simulation and Optimisation

7.1 Behaviour Simulation

7.1.1 Simulator Overview

In contrast to many teams in the legged league of Robocup, rUNSWift does not equate team cooperation solely with a passing game. Close quarter co-operation among the robots is integral to the behaviour of the team.[19] Doing extensive evaluation and testing on real robots are time consuming. This year rUNSWift started to look at some optimizations which requires evaluating large set of modules. Hence a behaviour simulator is developed.

By the nature of our goal, the simulator itself does not need to simulate every real aspect of the robot. For example, legs movement, most visual object recognition procedures are ignored, since they are of fewer interests in designing high-level behaviour modules. When building the simulator, our philosophy is to get the critical behaviour features simulated well with high priority. These features include, robot's geographic parameters such as position and heading, ball's parameters such as its position and velocity, the interaction between the robot and the ball, such as grabbing, dribbling and kicking, interaction between two objects such as collision between the ball and the robot and collision between robots.

7.1.2 Design

In real Robocup competition, each dog runs its separate copy of our code. In simulation, this is modeled by running each robot's behaviour on separate process. Our source code is there common for each process to import. At the backbone, there is a center process acting as a server. This process

is used to supervise each of the robot's behaviour, sort of like referee in a real game. This process is also in charge of updating user interface and is extended later to control the optimization procedures.

7.1.3 Implementation

The tool has undergone three different iterations. In its simplest form, the tool repeats a one-step simulation of the behaviour of a particular robot in many different positions on the field by feeding information as the position of the ball, teammates and opponents to the behaviour module. The result is visually displayed as a vector field and any unexpected behaviour can be quickly spotted.

Since many of our more complex behaviors are modeled as a more detailed state machine, the second iteration of this tool runs a complex behaviour over multiple frames, simulating a single agent's behaviour.

The third iteration of the tool runs each such simulation as a separate process, so that simulating a full 4 vs 4 game is feasible.

7.1.4 Plug-In-And-Play

Back to year 2004, rUNSWift has decided to port all its behaviour modules into Python. Nowadays, using a scripting language to specify behaviour is not new. Cross compile C++ and Python is not trivial but once done the benefits become apparent. [36][35][21] Since all behaviors are specified in Python, the kernel of this simulator is written also in Python. This allows us to change a few lines of code in one behaviour modules and re-run the simulator without any compilation delay.

Currently the simulator has interfaces in PyQt, the Python version of QT, and one version in OpenGL supporting multiple views.(7.1,7.2,7.3)

7.2 Behaviour Optimization

7.2.1 Evaluation

Evaluation methods for behaviour modules present a difficult problem of its own. There are large number modules to tune up, and qualities of these modules not only rely on the design of themselves but also the performance of the rest of the system. The goal of the behaviour system is to have one set

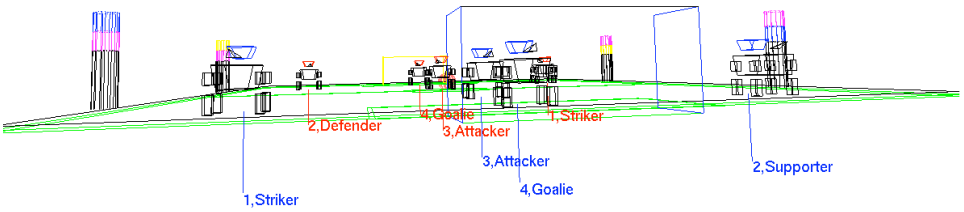

Figure 7.1: The simulattion of the ready state with both teams on the regular attacker, striker and defender strategy. The Red team is kicking off.

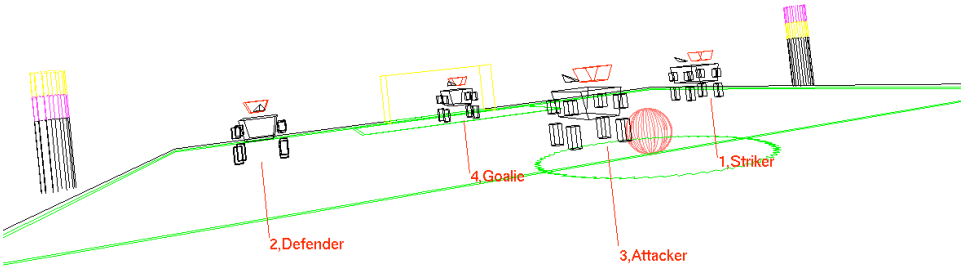

Figure 7.2: The simulator is simulating the same ready state in different view, a zoomed in view from an opponent robot's prospective.

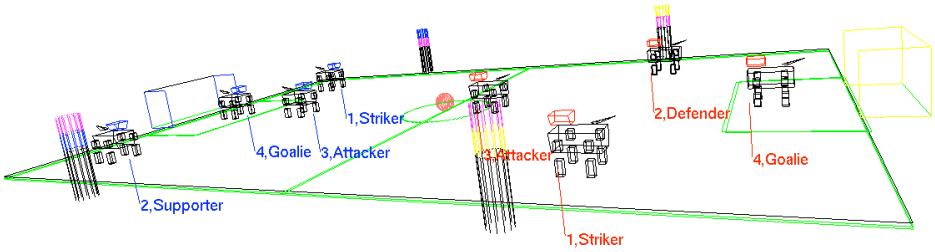

Figure 7.3: 2006 rUNSWift 3D Behaviour Simulator: Same ready state in yet another different view, an audiance's prospective

of behaviour modules consistently winning matches over all other possible ones by using equivalent vision, GPS, walk engine etc.

With behaviour simulator, one can perform some form of behaviour module evaluations, which is of the essential nature as above. In the simplest form, "evaluation" could mean a serials run of games on the simulator for several candidate modules versus a fixed opponent logic, in order to determine which of these candidate modules is likely to perform better.

In particular, if one set of behaviour logic consistently beat another set of modules, then it is likely that this set of module can in fact play a better game in the real.

7.2.2 Optimization

Optimization is an integral part of module evaluations. Where evaluation picks the best module among a set of candidates, optimization improves one function/module further on its quality. Optimization on the simulator could usually provide good indications and hints on real robots with much less time consumption. The most straightforward application of this idea is to optimizing constants used in our behaviour modules. The input of our optimization procedure is a vector of constants, most of those are magic numbers we come up ourselves. The procedure goes on and applies function minimizations over this vector with one of the popular algorithms. The evaluation function is simply given by the score difference between two teams each running with different such constant vector. Also, to reduce noise and over fitting, single step evaluation should be conducted with sufficient time duration. (We typically use 10 minutes game result on the simulator) Finally, the larger the score difference is, the more biased we are towards to the function/module that results the higher score.

To validate as the last step, we run games on the simulator between logic that uses the vector we start with, as well as the vector outputted by our function minimization algorithm. The result of each game is recorded and finally the conclusion can be made based on these results. In an experiment we ran 10 validation simulations between the optimized data and the original data. The scores difference between optimized data set and original data set are given by:

Game	1	2	3	4	5	6	7	8	9	10
Difference	307	236	-86	88	-22	88	180	187	-28	165

Assume the difference between the optimized data set and original data set is a random variable Z that has a normal distribution.

$$Z \sim (\mu, \alpha^2) \tag{7.1}$$

The model hypothesis is given by:

$$\mu > 0$$

And the Null hypothesis is:

$$\mu \leq 0$$

The reject region used here is:

$$W_0 = \mid t \mid > 2.262, and, P > 0.95 \tag{7.2}$$

The formula is given by:

$$t = \frac{\sqrt{t} \times Z}{S} \tag{7.3}$$

From the results of the simulation, $n = 10$, $Z = 55.75$, and $S = 63.32$. T is calculated to be:

$$T = 2.78 > 2.262 \tag{7.4}$$

Therefore the resulting data supports our model hypothesis that $\mu > 0$. This means the our optimized vector indeed improved the performance in terms of scoring. In practice one will need to test the result in real games before making the final judgment.

7.3 Summary

Behaviour simulation made it possible to instantly see the result of using newly developed behaviour module. This allows us to quickly fix those bugs and unexpected behaviors before testing on real robots. Simulation runs much faster than real game, hence it also allow us to compare two set of behaviour modules by simulating the game in a long period. For example over night. The team has also conducted experiment on optimizing behaviors by applying function minimization on simulations. However, the team feels less confident on these kind of automatic approaches and in practical the optimized functions are used as a guide only, final decision is usually made upon experiments on the real robots.

Chapter 8

Conclusion and Outlooks

8.1 Conclusion

Standing at this point in time, looking backwards, we feel rUNSWift has done overally well in year 2006. The team has come 1st in Australian Open and 2nd in the World Championship. Great improvements were developed by the team in GPS, Vision and Behaviors.

In the context of vision modules, rUNSWift has performed further upgrade of its vision system this year moving from Ripple Down Rules[38],[39] based classifier used in 2005 to a new kernel based classification system. The speed, accuracy and reliability of our vision system is further maintained at high quality this year. Optimizations on calibrations, linear shift, ring correction, field edge recognition and sanity checks were conducted. [1]

In the domain of behaviour system, the team emphasizes the grab-dodge-shoot philosophy this year. Enormous amount of time were spent on the four critical modules: find ball, tracking, grabbing, and grab-dribble, plus Role positioning and role assignment. The entire behaviour module has demonstrated its incremental performance through Australian Open to the World Final.

8.2 Outlook

Standing at this point in time, looking backwards, we also realized that there are many areas where possibilities of improvements remained. Behaviour is the most volatile part in Robocup four-legged league. Continuing development is certainly needed for next year's Robocuppers. Among those ideas, one can investigate the possibility of ball passing, this includes should both

[1] For detailed description of the vision system please refer to Andy Owen's Thesis

deliberate and opportunistic passing. In addition, the declarative strategy used in role positioning can be further elaborated in other area in behaviour level. Furthermore, development of a quick and reliable kick in both long range and short distance is also not a bad choice.

At the very last, I hope the layout of this thesis is clear, and the concepts and ideas presented here would still be useful for the upcoming year's Robocup enthusiasts in the four-legged league.

Acknowledgements

First of all I would like to thank everyone on the team, Michael, Andy, Oleg, Ryan. It is the teamwork and team cooperation that boosted our performance. I would also like to thank Brad Hall, for his guide and advice from the very beginning and all the way to the end. Thanks to Professor Claude's advices and suggestions. Thanks to Dr Waleed for letting me use Leo cluster. Special Thanks to Nobuyuki - His early framework on behaviour modules is a perfect start for my work this year. And last but not least, my supervisor, Dr William Uther, Thanks very much for believing in us, and making us who we are.

Life is always full of challenges, and I won't forget those hard working days with all of you. Thanks.

Enyang HUANG, 1, September, 2006

Bibliography

[1] Andrew Owen *The 6^{th} Sense I See Red People (As Ball)*, rUNSWift Undergraduate Thesis, University of New South Wales, 2006

[2] Oleg Shushkov *Robot Localizaion Using a Distributed Multi-Modal Kalman Filter, and Friends*, rUNSWift Undergraduate Thesis, University of New South Wales, 2006

[3] Ryan Cassar *Four Legs and A Camera*, rUNSWift Undergraduate Thesis, University of New South Wales, 2006

[4] Alex North, *Object Recognition From Sub-Sampled Image Processing*, http://www.cse.unsw.edu.au/ robocup/2005site/ reports05/north05-subsampled.pdf, 2005, University of New South Wales

[5] S.B.Kang and R.S.Weiss, *Can We Calibrate a Camera Using an Image of A Flat Textureless Labertian Surface*, 2000.

[6] H.Nanda and R.Cutler, *Practical Calibrations for a Real-Time Digital Omnidirectional Camera. Technical report*, CVPR 2001 Technical Sketch, 2001.

[7] Walter Nistic, Thomas Roger *Improving Percept Reliability in the Sony Four-Legged League*, Robocup, 2005.

[8] Xu, Jing, *2004 rUNSWift Thesis - Low Level Vision*, rUNSWift, 2004.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.*Introduction to Algorithms*, Second Edition. MIT Press and MaGraw-Hill, 2001. Section 29.3, pp. 790-804.

[10] S. Kirkpatrick and C.D. Gelatt and M.P.Vecchi,*Optimization by Simulatted Annealing*, Science, Vol 220, Number 4598, pp 671-680, 1983, http://citeseer.ist.psu.edu/kirkpatrick83optimization.html

[11] Nobuyuki, Morioka *Behaviour Module Design and Implementation, System Integration*, Thesis Report, Univerisity of New South Wales, 2005. http://www.cse.unsw.edu.au/.../thesis_nobuyuki.pdf

[12] Jared Bunting, Stephan Chalup, Michaela Freeston, Will McMahan, Rick Middleton, Craig Murch, Michael Quinlan, Christopher Seysener, and Graham Shanks. *Return of the NUbots: The 2003 NUbots Team Report*, Technical Report, Newcastle Robotics Laborotory, The University of Newcastle, 2003.

[13] M.A.Fischler and R.C.Colles (June 1981) *Random Sample Consensus: A Paradim for Model Fitting with Applications to Image Analysis and Automated Cartography*, Comm. of the ACM 24: 381–395

[14] Chen FX, Wang RS *Fast RANSAC with preview model parameters evaluation*, Jornal of Software, 2005, 16(8): 1431-1437

[15] R.A.Fisher*The Goodness of Fit of Regrssion Formulae, and The Distribution of Regression Coefficients*, J.Royal Statist. Soc., 85, 597-612, 1922.

[16] Wikipedia, 2006 *The Generic Ransac Algorithms*
http://en.wikipedia.org/wiki/RANSAC, 2006

[17] S. Chen, M. Siu, T. Vogelgesang, T. F. Yik, B. Hengst, S. B. Pham, C. Sammut. *The UNSW RoboCup 2001 Sony Legged League Team.* Technical Report, The University of New South Wales.

[18] Z. Wang, J. Wong, T. Tam, B. Leung, M. S. Kim, J. Brooks, A. Chang, N. V. Huben. *UNSW RoboCup 2002 Sony Legged League Team.* Bachelor Thesis, The University of New South Wales.

[19] J. Chen, E. Chung, R. Edwards, N. Wong. *Rise of the AIBOs III - AIBO Revolutions.* Bachelor Thesis, The University of New South Wales and National ICT Australia.

[20] Thomas Rofer, Tim Laue, Han-Dieter Burhard, Jan Hoffmann, Matthias Jungel Thoman Rofer, Tim Laue, Hans-Dieter Burkhard, Jan Hoffmann, Matthias Jungel abd Daniel Gohring, Martin Lotzsch, Uwe Duffert, Michael Spranger, Benjamin Altmeyer, Vi- viana Goetzke, Oskar von Stryk, Ronnie Brunn, Marc Dassler, Michael Kunz, Max Risler, Maximilian Stelzer, Dirk Thomas, Stefan Uhrig, Uwe Schweigelshohn, Ingo Dahm, Matthias Hebbel, Walter Nistico, Carsten Schumann, and Michael Wachter. *GermanTeam 2004. Technical Report*, University at Bremen, Humboldt-University at zu Berlin, Technische Univer- sity at Darmstadt, University of Dortmund, 2004.

[21] Micheal J.Quinlan, Steven P.Nicklin, Kenny Hong, Naomi Henderson, Stephen R.Young, Timothy G.Moore, Robin Fisher, Phavanna Douangboupha, Stephan K.Chalup, Richard H.Middleton and

Robert King *The 2005 NUbots Team Report*, 2005, Univerisity of Newcastle

[22] J. Shammay. *Real-Time Shared Obstacle Probability Grid Mapping and Avoidance for Mobile Swarms* Bachelor Thesis, The University of New South Wales and National ICT Australia.

[23] Christopher G. Atkson, Andrew W. Moore, ans Stefan Schaal. *Locally Weighted Learning*, October 12, 1996.

[24] Mayers, R.H. (1990). Classical and Modern Regression With Applications. PWS-KENT, Boston, MA.

[25] Press, W.H., Teukoslsky, S.A., Vetterling, W.T., and Flannery, B.P.(1988). *Numerical Recipes in C.* Cambridge University Press, New York, NY.

[26] Gasser,T. and Muller, H.G (1979) *Kernal Estimation of Regrssion Functions.*, page 23-67. Springer-Verlag, Heidellberg.

[27] Deheuvels,P *Estimation Non-parametrique del la Densite Par Histogrammes Gerneralises.* Revue Statistique Applique, 25:5-42

[28] Wand, M, P and Schucany, W.R *Gaussian-Based Kernels for Curve Estimation and Window Width Selection.* Canadian Journal of Statistics, 18:197-204

[29] Wikipedia, 2006 *The 3-demensional kd-tree* http://en.wikipedia.org/wiki/Kd-tree, 2006

[30] Bentley, J.L *Multidimensional Binary Search Trees Used for Associative Searching.* Communications of the ACM, 18(9):509-517

[31] Friedman, J.H, Bentley, J.L, and Finkel, R.A*An Algorithm for Finding Best Matches in Logarithmic Expected Time.* ACM Transactions on Mathematical Software, 3(3):209-226

[32] Samet, H *The Design and Analysis of Spatial Data Structures* Addison-Wesley, Reading, MA.

[33] Sproull, R.F *Refinements to Nearest-Neighbor Seaeching in K-D Trees.* Algorithmica, 6:579-589.

[34] Tou, J.T and Gonzalez, R.C *Patern Recognition Principles* Addison-Wesley, Reading, MA.

[35] Hayato Kobayashi, Tsugutoyo Osaki, Akira Ishino, Jun Inoue, Narumichi Sakai, Satoshi Abe, Shuhei Yanagimachi, Tetsuro Okuyama, Akihiro Kamiya, Kazuyuki Narisawa, and Ayumi Shinohara *Jolly Pochie Technical Report Robocup 2005*, 2005

[36] Gilad Buchman, David Cohen, Paul Vernaza, and Daniel D.Lee *Univerity of Pennsylvania Robocup 2005 Technical Report*, 2005

[37] Schaal, S. and Atkeson, C.G *Assessing the Quality of Learned Local Models.* In Cowan et al. (1994), page 160-167

[38] Compton, P., Peters, L., Edwards, G., and Lavers, T.G.*Experience with Ripple-Down Rules*, Knowledge-Based System Journal, 2006.

[39] Compton, P. J. and R. Jansen *A Philosophical Basis for Knowledge Acquisition.* Knowledge Acquisition 2: 241-257, 1990

[40] Xiaohan Z., Kai X., Xiaohui L., Fei L., Hao Z., Bo R., Xiaoping C. *WrightEagle 2005 Legged Team Report*, Technical Report, University of Science and Technology of China, 2005

[41] Thomas Rofer, Tim Laue, Michael Weber,Hans-Dieter Burkhard, Matthias Jungel, Daniel Gohring, Jan Hoffmann, Benjamin Altmeyer, Thomas Krause, Michael Spranger, Oskar von Stryk, Ronnie Brunn, Marc Dassler, Michael Kunz, Tobias Oberlies, Max Risler, Uwe Schwiegelshohn, Matthias Hebbel, Walter Nistico, Stefan Czarnetzki, Thorsten Kerkhof, Matthias Meyer, Carsten Rohde, Bastian Schmitz, Michael Wachter, Tobias Wegner, and Christine Zarges *German Team Robocup 2005*, Technical Report, University at Bremen, Humboldt-University at zu Berlin, Technische University at Darmstadt, University of Dortmund, 2005