THE UNIVERSITY OF NEW SOUTH WALES SCHOOL OF COMPUTER SCIENCE & ENGINEERING

The Sony Legged Robot League
COMP4911 Thesis B report

Author: Ted Wong (2274294)
Computing Engineering

Supervisor: Will Uther

Assessor: Claude Sammut

Submission: 13.9.2004

Contents

Chapter 1
Introduction

1.1: Overview

rUNSWift is a robotic soccer team established by UNSW. In 2004, rUNSWift participated in the Sony Four-Legged Robot League in Lisbon, Portugal and was one of the 1/4 finalists. This report is a technical report of my RoboCup thesis. In this report, the rUNSWift software system, theories, experimental results will be described in details.

This report is divided into 5 chapters: *Introduction, Vision, Wireless and locomotion, Behavior before Australian Open, Behavior After Australian Open.* The first chapter is an overview of this report. The second chapter describes the robot's vision. The next chapter describes the robot's wireless and locomotion module. The fourth chapter describes the behavior module before the 2004 RoboCup Australian Open competition. The final chapter describes the behaviors after the competition.

1.2: Sony Four-Legged Robot League

Sony Four-Legged Robot League [1] is part of the RoboCup competition. RoboCup is an international artificial intelligence robotic soccer competition. The ultimate goal of RoboCup is *"By 2050, develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer. [2]*" . The technologies invented for RoboCup can be applied to our society.

In the Sony Four-Legged Robot League, robotic dogs are used to play soccer matches. There are other leagues including: Middle Size League, Small Size League, Humanoids League, Simulation League, Rescue League, E-League and Junior League. Interested readers are referred to [2].

Each Four-Legged League soccer match consists of two ten minutes halve. Four robots in each team with one goalie. The winner

is the team who has scored the most goal. The soccer field is 4x2m wide, it has one blue goal, one yellow goal and four beacons. Wireless communication between the robots are allowed during the match. All robots are autonomous, human interference is not allowed. All the teams must use the Sony robots, physical modifications are not allowed.

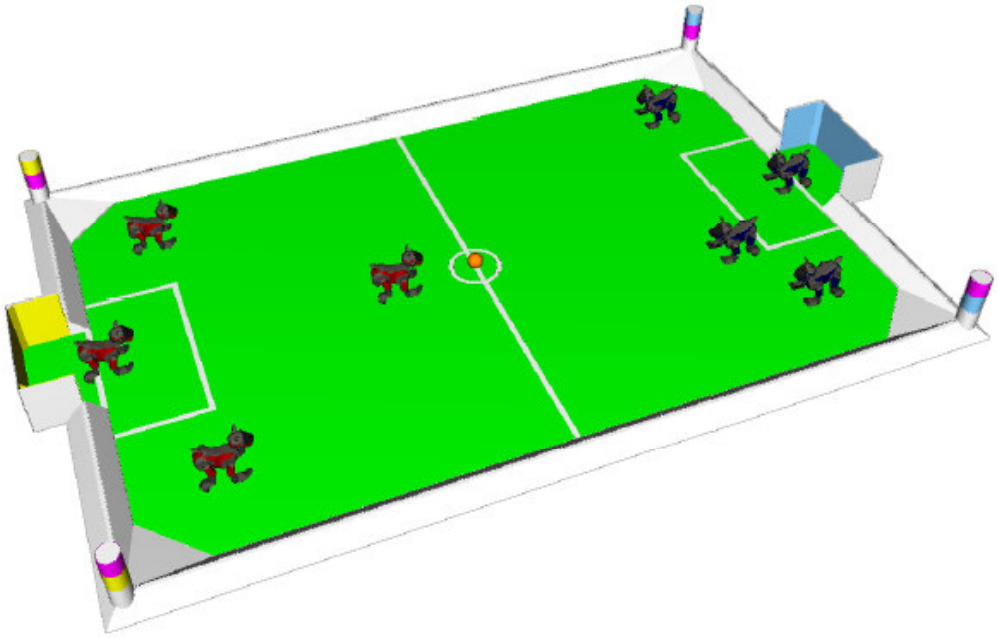


Figure 1.1: Sony Four-Legged Robot League soccer field. (Image courtesy of Sony Four Legged Robot Football League Rule book). [3]

Aiming to beat the FIFA world cup team by 2050, this year the Sony Four-Legged League committee has modified the rules, making the games more challenging than the previous years.

Rule modifications [4]:

- The border height has been reduced to 30cm. Cutting the border height present vision challenges.

- This year any robot charging for another robot for over three seconds would be taken off the field for 30 seconds, unless it is the robot closest to the ball with no more than one robot length apart.

- The two middle beacons have been removed. The robots have less

information about their position.

- This year all the robots are expected to reposition themselves. Previously this was done manually.

1.3: Sony AIBO robots

In this year's competition, two Sony AIBO models — ERS-7 and ERS-210 were participating. Previously rUNSWift used the ERS-210 robots, this year the team decided to switch to ERS-7. Refer to [10] for more details on the ERS-210 model.

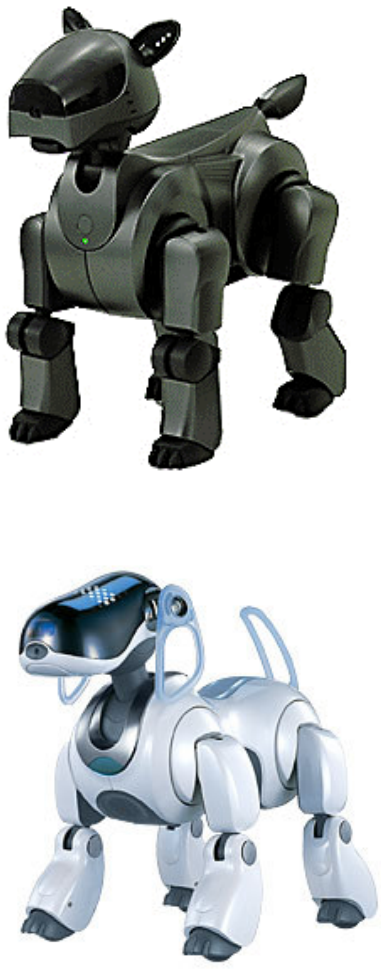


Figure 1.2: Sony ERS-210 robot (top)
and ERS-7 robot (bottom)

The ERS-7 robot has four legs. It has a camera, its resolution is 416 X 320 [11]. It processes 30 camera frames per second. The

robot has three distance sensors — chest sensor, head near sensor and head far sensor. Chest sensor allows the robot to detect obstacles in front of the chest. The near sensor detects obstacles with distance between 5cm to 50cm. The far sensor detects obstacles with distance between 20cm to 150cm [11]. The robot's head has three degrees of freedom — pan, tilt and crane.

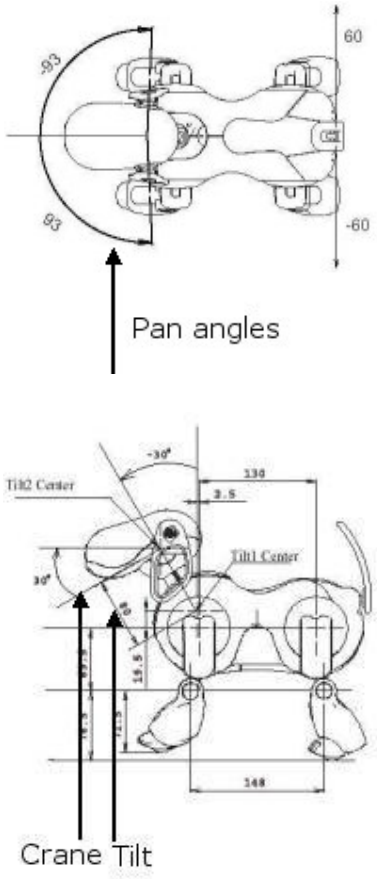


Figure 1.3: Pan, tilt and crane. (Images courtesy of Sony OPEN-R SDK Model Information for ERS-7). [11]

rUNSWift decided to switch to the ERS-7 robots because:

- ERS-7 robots have a higher camera resolution. Hence the ERS-7 robots can see the far objects (eg: far beacons) better than the old robots.
- ERS-7 has stronger motors, they can move faster and hit the ball further than the old model.

- ERS-7's head has three degrees of freedom, the old model has only two degrees. Higher degrees of freedom allows smooth head movement.

- ERS-7 has a more powerful CPU than the old models. Its frame rate is 30 frames per second while the ERS-210 can only process 25 frames per second. Faster frame rate allows the software to send more commands to the hardware.

Major physical differences between the ERS-7 and ERS-210 are discussed in chapter 4.

1.4: Software Architecture

1.4.1: 2003 software architecture

In 2003, rUNSWift software system was broken into five modules, namely vision, localisation, locomotion, behavior and wireless.

Vision

The vision module processes camera images and classify them into color pixels. Neighbor pixels of the same color are joined together to form a blob. Some of these blobs are recognized as objects on the field. Recognized objects can be used in the higher level modules — localisation and behavior.

Localisation

The localisation module receives information from the vision, wireless and locomotion module. The information is analyzed and form a world model. A world model is an internal representation of a soccer field, it contains the robot's and other robot's current location and the ball location. Camera images provide useful information about the robot's self position, known as vision update. locomotion odometer tells how far the robot has moved since the last update, known as motion update. Information related to the teammates are received from the wireless.

This year, a field edge detection algorithm allowing the robots to localise with the field lines was invented by [7].

Locomotion

The locomotion module receives action instructions from the behavior module, interpret and forward them to the hardware.

Behavior

This is the highest level module. The robot decides what , when and where to do. The decision is depend on number of factors such as the world model and joint values.

Wireless

The wireless module sends and receives wireless packets from/to the teammates. It also receives wireless commands from the basestation [1] and the gamecontroller[2].

All the modules were written in C++.

Wireless, locomotion and vision module need to interact with the hardware. This is done with OPEN-R [12]. OPEN-R is an application programmer interface for the Sony robots. OPEN-R provides a set of functions for the rUNSWift softwares to use and interact with the hardware.

rUNSWift system was implemented by three OPEN-R objects. Each object interacted with other objects and the environment through the OPEN-R. The vision object included the behavior, localisation and vision module. The actuatorcontrol object contained the locomotion module. The wireless object contained the wireless module. Information from the vision module passed to the localisation and behavior module, information from the localisation passed only to the behavior module. The behavior module sent/received wireless teammate info through the wireless module. It was the responsibility of the wireless module to convert the wireless teammate info to wireless packets and also the other way

1A PC program communicate with the robots.
2A program allowing the robots to play a soccer match.

around. The actions performed by the robot were sent from the behavior to the locomotion. The locomotion module converted the robot actions to motor positions.

The vision module received camera images from the camera sensor. The wireless module sent/received wireless packets through the OPEN-R TCP gateway. The locomotion module received odometer update from the leg sensors. The robot performed its actions through the leg effectors.

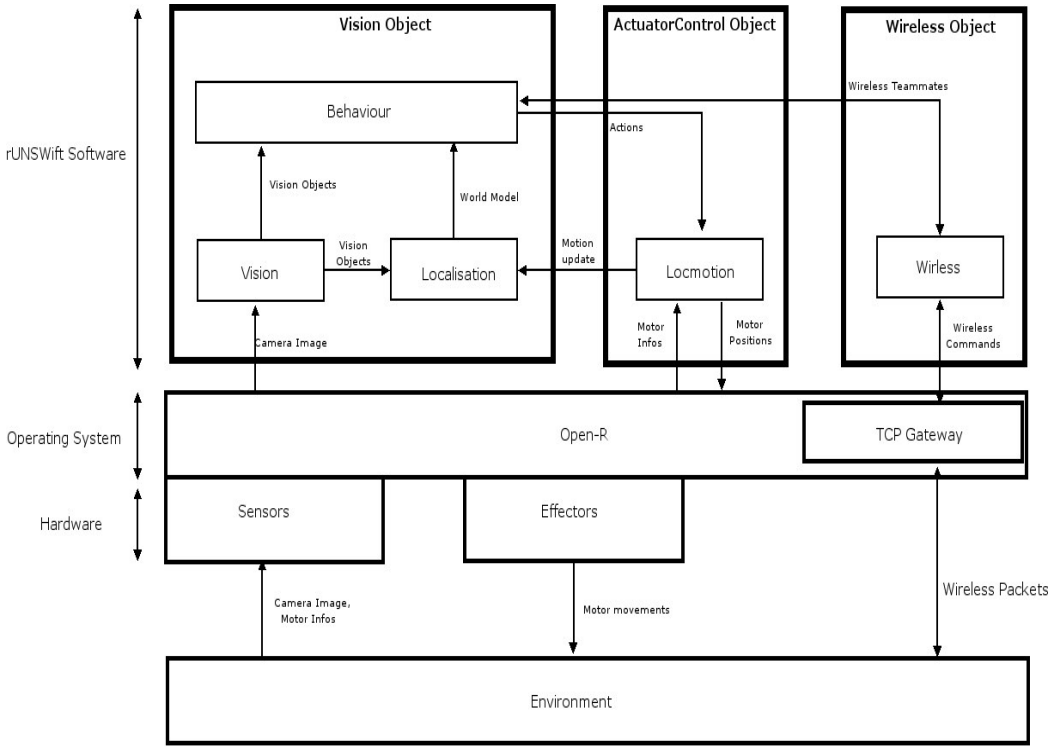


Figure 1.4: 2003 rUNSWift software architecture.

1.4.2: 2004 software architecture

rUNSWift's current software architecture is similar as the previous year. However the current system is composed of two programming languages, C++ and Python. The low levels (wireless, localisation, locomotion and vision) are still written in C++. The high level (behavior) is now written both in C++ and Python. Python offers a number of advantages over C++.

Most of the high level codes are now written in Python. Not

all the behavior codes are ported into Python due to limited development time. rUNSWift started to port the behavior from C++ to Python after the Australian Open, two months before the world open begun.

Python behaviors receive vision, wireless, locomotion and localisation information from the C++ modules. After it processes the information, it sends the action instructions and wireless commands back to the C++ modules. Refer to section 5.4 for more details.

This year rUNSWift had decided to switch from TCP wireless to UDP wireless, refer to section 3.4.1 for more details.

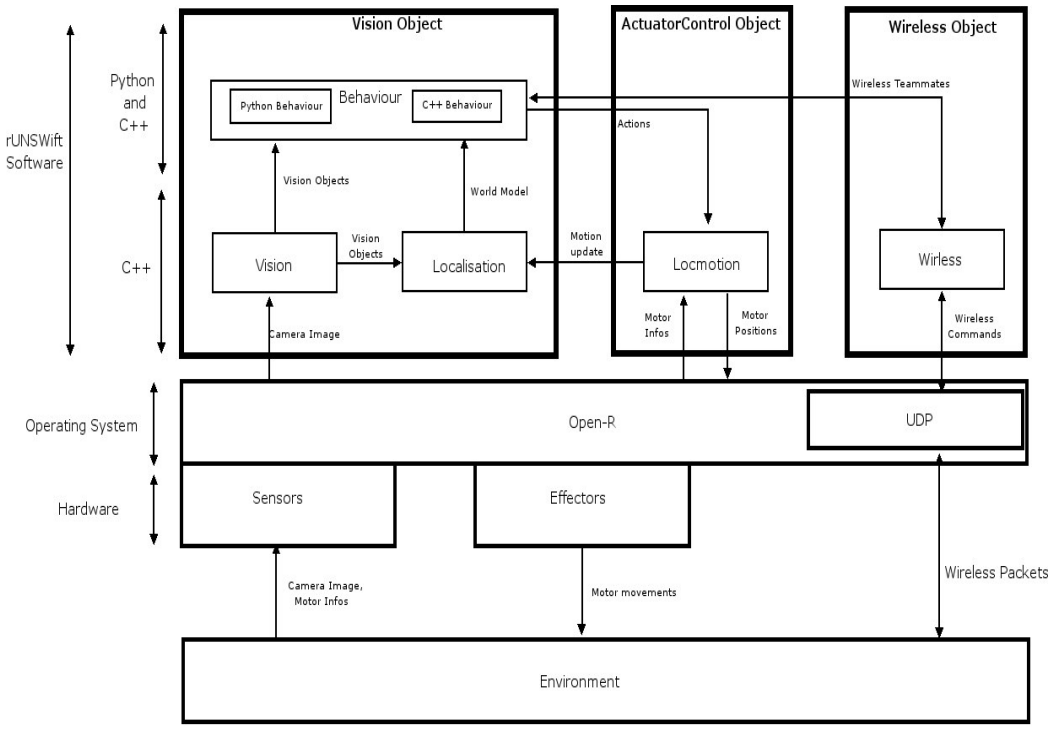


Figure 1.5: 2004 rUNSWift software architecture.

1.5: rUNSWift software terminologies

This section describes rUNSWift's terminologies briefly. The terminologies are used in the later chapters.

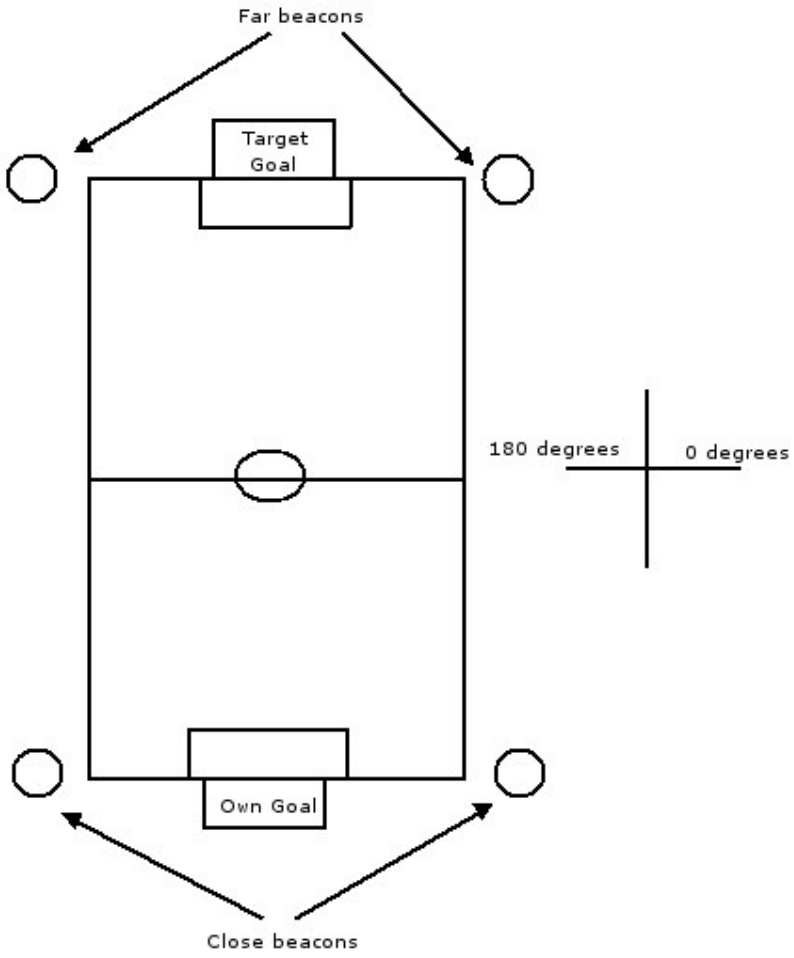


Figure 1.6: Global heading and beacons.

The direction a robot is facing is known as heading. Global heading is relative to the soccer field, zero global heading is the direction pointing toward the right hand side of the field as depicted in figure 1.6. The two beacons closest to the own goal are known as close beacons. The two beacons closest to the target goal are known as far beacons.

Chapter 2
Vision

2.1: Introduction

rUNSWift robots capture images known as cplane from their camera. The images taken contain a number of pixels. Each pixel has a YUV component. The YUV component is translated to a color by a color table. Hence each pixel is associated with a color. There are 13 different colors:

- Orange
- Beacon blue
- Beacon green
- Beacon yellow
- Beacon pink
- Robot blue
- Robot red
- Field green
- Robot grey
- White
- Black
- Field line
- Field border

Vertically connected identical color pixels are joined and formed as a segment, this process is known as color segmentation. The segments are then passed to the blob algorithm. The role of the blob algorithm is to merge identical color segments horizontally, known as blob. The output of the blob algorithm is a number of blobs.

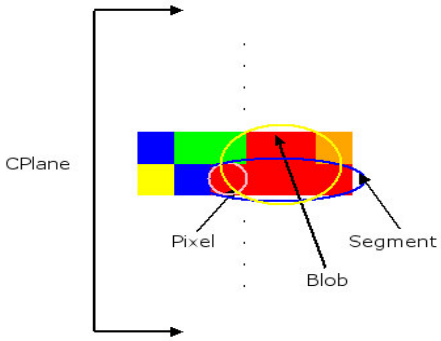


Figure 2.1: Pixels, segments, blobs and CPlane.

The blobs are recognized as objects on the field by the object recognition. Unfortunately sometimes the object recognition recognizes fake objects, they must be discarded, this is done in the sanity check module. Objects which have passed the sanity check module can be used in the localisation and behavior.

The details of the color calibration, object recognition and sanity check can be found in [5] and [8]. In the following sections, the blob algorithm is described in details.

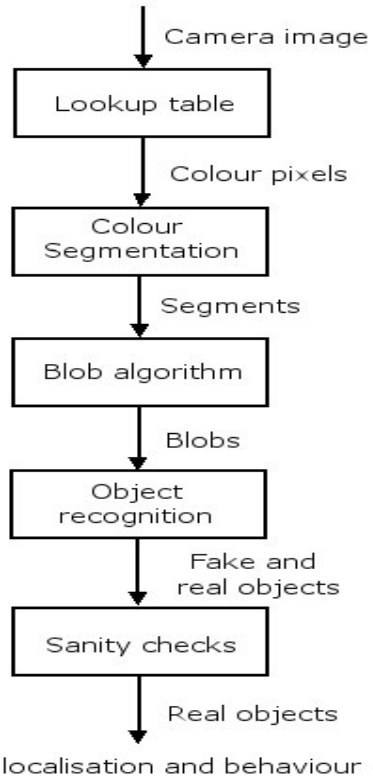


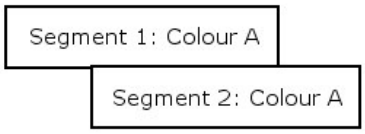
Figure 2.2: Block diagram of the vision module

2.2: 2003 blob algorithm

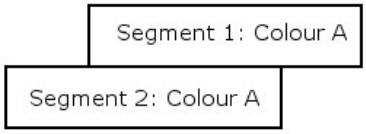
In the 2003 rUNSWrift report the students stated that the blob algorithm was buggy (page 26 of [13]). They believed the algorithm was incorrect because sometimes not all the connected orange pixels were recognized as a orange ball.

However this is a misunderstanding, the blob algorithm was correct but some minor rounding off errors on the cplane made the blob display on the cplane look incorrect. The bug has been fixed. Furthermore a new blob algorithm has been written, replacing the blob old algorithm.

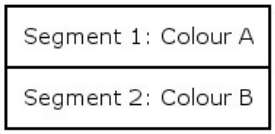
The old blob algorithm scans segments from the top to bottom row, left to right each row. During the scan the algorithm compares each segment with all the segments on the previous row and merge them if they overlap. Overlap occurs when two identical color segments are connected vertically, see figure 2.3.



Segment 1 and 2 are overlapping. They are connected and they have the same colour.



Segment 1 and 2 are overlapping. They are connected and they have the same colour.



Segment 1 and 2 are not overlapping. They are connected but they don't have the same colour.

Figure 2.3: Overlap examples

If the segment being scanned has no overlap, no action is taken. If it has an overlap, then:

- Overlap case one: It is not joined previously
- Overlap case two: It is joined previously

Lets take an example, look at figure 2.4 for the first overlap case. Three segments on row y – 1 are connected with segment A. However only one segment — segment B has the same color as segment A. Hence the blob algorithm merges segment A with blob BC. Segment A is not joined with any other segments previously.

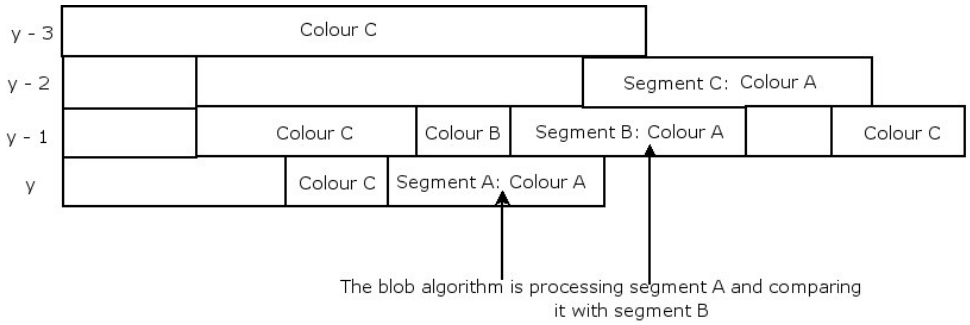


Figure 2.4: Case one. Segment A joins with segment B. Segment A hasn't joined with any other segment previously.

Lets take an example of the second overlap case. Assume we have a segment plane like in figure 2.5. Since the blob algorithm scans segments from left to right, the algorithm joins segment A and C before it joins segment A and B.

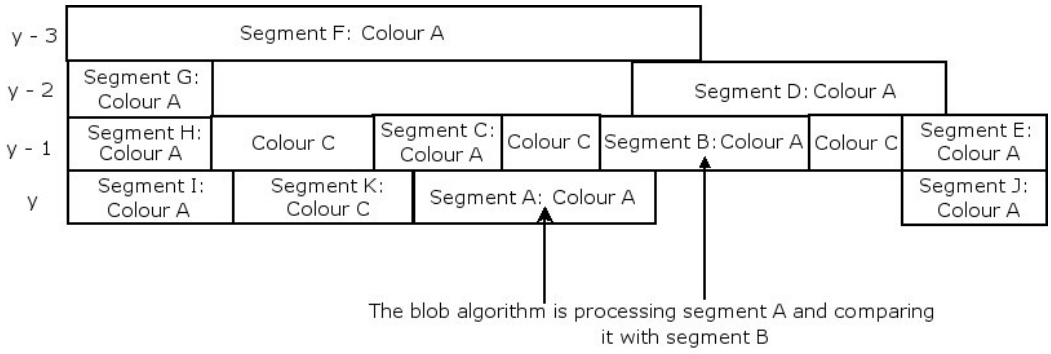


Figure 2.5: Case two. Segment A joins with segment B. Before they are merged, segment A has already mergd with segment C since the blob algorithm scans segments from left to right.

The merging process for the first overlap case is simple and fast. The blob algorithm simply joins the two overlapped segments. For example in figure 2.4, the algorithm joins segment A and B and update their information such as the blob number. Only segment A

and B need to be processed.

However the merging process for the second overlap case is more complicate and slow. The algorithm has to go through all the segments on the previous row beginning with the overlapped previous row segment. The algorithm also needs to go through all the segments on the current scanning row starting from the first segment to the segment before the segment being scanned. For example in figure 2.5, before segment A and B are joined, segment B is already joined with segment D, E, F, G, H and I - blob BDEFGHI. The blob algorithm firstly needs to scan all segments from segment B to E, in the example three segments need to scan. Since segment B and E are parts of the blob BDEFGHI, they are joined with segment A and their information is updated accordingly. Segment E must be processed because there may be a color A segment such as segment J which overlaps with segment E. Segment J must be able to merged with segment A when the algorithm merges segment E and J. The next step is scan all the segments from segment I to K, in the example two segments (I and K) need to scan. Since segment I belongs to the blob BDEFGHI, it is joined with segment A. Overall the merging process takes five comparisons.

This algorithm always produce the correct result but it is slow. The algorithm is slow because it needs to scan many segments during the second overlap merging, some of those are scanned multiple times. The worse running time occurs when the camera image is noisy. The algorithm needs to scan many small and meaningless pixels. The running time for the first overlap case is O(1), the second overlap case is O(w), and the overall algorithm running time is hwO(w), where h is the height of the cplane and w is the width of the cplane.

```
/*
   Copyright 2004 The University of New South Wales (UNSW) and
   National ICT Australia (NICTA). This file is part of the 2004
   team rUNSWift RoboCup entry. You may redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of
   the License, or (at your option) any later version as modified
   below.  As the original licensors, we add the followin
   conditions to that license: In paragraph 2.b), the phrase
   "distribute or publish" should be interpreted to include entry
   into a competition, and hence the source of any derived work
   entered into a competition must be made available to all parties
   involved in that competition under the terms of this license.
   In addition, if the authors of a derived work publish any
   conference proceedings, journal articles or other academic
   papers describing that derived work, then appropriate academic
   citations to the original work must be included in that
   publication. This rUNSWift source is distributed in the hope
   that it will be useful, but WITHOUT ANY WARRANTY; without even
   the implied warranty of MERCHANTABILITY or FITNESS FOR A
   PARTICULAR PURPOSE.  See the GNU General Public License for more
   details. You should have received a copy of the GNU General
   Public License along with this source code; if not, write to the
   Free Software Foundation, Inc., 59 Temple Place, Suite 330,
   Boston, MA  02111-1307  USA
*/

void form2003BlobAlgorithm() {

      ......

      // Scan all the segments from top to bottom, and left to right.
      for (int y = 1; y < CPLANE_HEIGHT; y++; prev_y) {

            // w is the index of previous row segments.
            int w = 0;

            // x is the index of current row segments.
            for (int x = 0; x < segmentCount[y]; x++) {

                  // cs is the current segment.
                  cs  = splane.segment[y][x]

                  for (; w < segmentCount[prev_y]; w++) {

                        // ps is previous segment.
                        ps = &splane.segment[prev_y][w];

                        // Overlap occurs if the segments have  the same
                        // color and they are connected.
                        if (cs->color == ps->color &&  ps->connect(cs)) {

                              // Overlap case one.
                              if (cs->blob_number == -1) {
                                    cs->blob_number = ps->blob_number;
                                    blobinfo[ps->color][ps->blob_number].
                                    Update(cs->startIndex, cs->endIndex,
```

```
                        y, cs->length, cs->xsum,
                        cs->ysum);
                }

                // Overlap case two.
                else {
                        updateBlob(splane.segment[prev_y], w
                            , segmentCount[prev_y] -
                            1);

                        updateBlob(splane.segment[y], 0, x);
                }

        }

        ......

}
```

Figure 2.6: A simplified code fragment of the 2003 blob algorithm.

2.3: Disjoint sets blob algorithm

Although the old blob algorithm is slow, its speed is still acceptable for the ERS-210, unfortunately this is not the case for ERS-7. ERS-7 has a higher camera resolution. Higher camera resolution implies the blob algorithm need to process more segments since the cplane contains more pixels. Also ERS-7 camera images are usually noisy, these small noisy pixels have a significant impact on the overall robot's processing power. Since blob algorithm is the most time consuming process, if it runs slowly then the robot may drops vision frames. Also the fact that ERS-7 has a faster frame rate (30 frames per second for ERS-7, 25 frames per second for ERS-210) suggest that a new blob algorithm must be developed. In fact rUNSWift ERS-7 robots encountered frame drops with the old blob algorithm, usually one or two frames drop per second.

This year a new blob algorithm known as disjoint sets has been written. The new algorithm receives the same inputs as the old algorithm and they both generate exactly the same outputs. However the disjoint sets blob algorithm runs much faster than the old algorithm since the running time for the second overlap case merging is minimised.

The idea of the disjoint sets algorithm comes from the disjoint sets algorithm theory [14]. Disjoint set data structure maintains a collection S = {S1,S2,...,Sk} of disjoint sets. Each

set is represented by a representative (root), which is also a member of the set. It doesn't matter which member is used as the representative, but only one representative is allowed. Two common disjoint sets implementation are link list and rooted trees. The disjoint sets blob algorithm only uses the rooted trees. Sets are represented by rooted trees with each node containing one member and each tree representing one set. This implementation also required that each node points only to its parent and each node is contained by only one disjoint set. The root of the tree contains the representative and is its own parent. See figure 2.7.

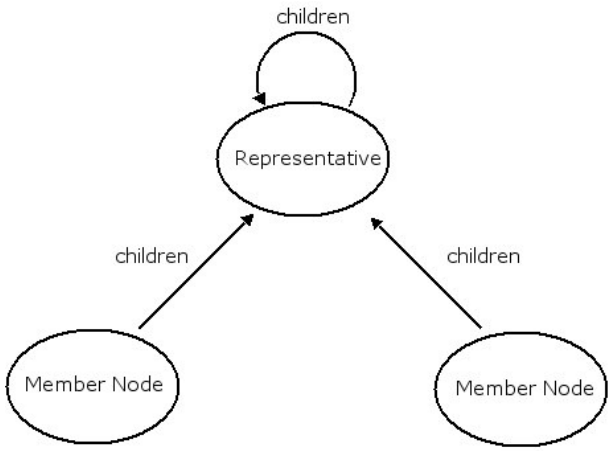


Figure 2.7: Disjoint sets. Each set points to its parent. Representative's parent is itself.

There are three common disjoint sets operations:

- *Make a set*

  Create a new set with only one node. That node is also the representative of the set. It points to itself.

- *Union two sets*

  This operation unites two sets. The new set is the union of these two sets. Lets call these two sets as S1 and S2. The representative of the union set is the representative of S1 or S2 based on path compression heuristic. See figure 2.8.

- *Find set representative*

  Given a node, this operation returns a pointer to the representative of the set containing the given node. Path compression is applied. The running time of the tree is correspond to its rank — lower rank translates to smaller running time. The idea of path compression is to make each node on the find path point directly to the root, hence reduce the rank and "*speed-up any subsequent queries on the find path*". [15].

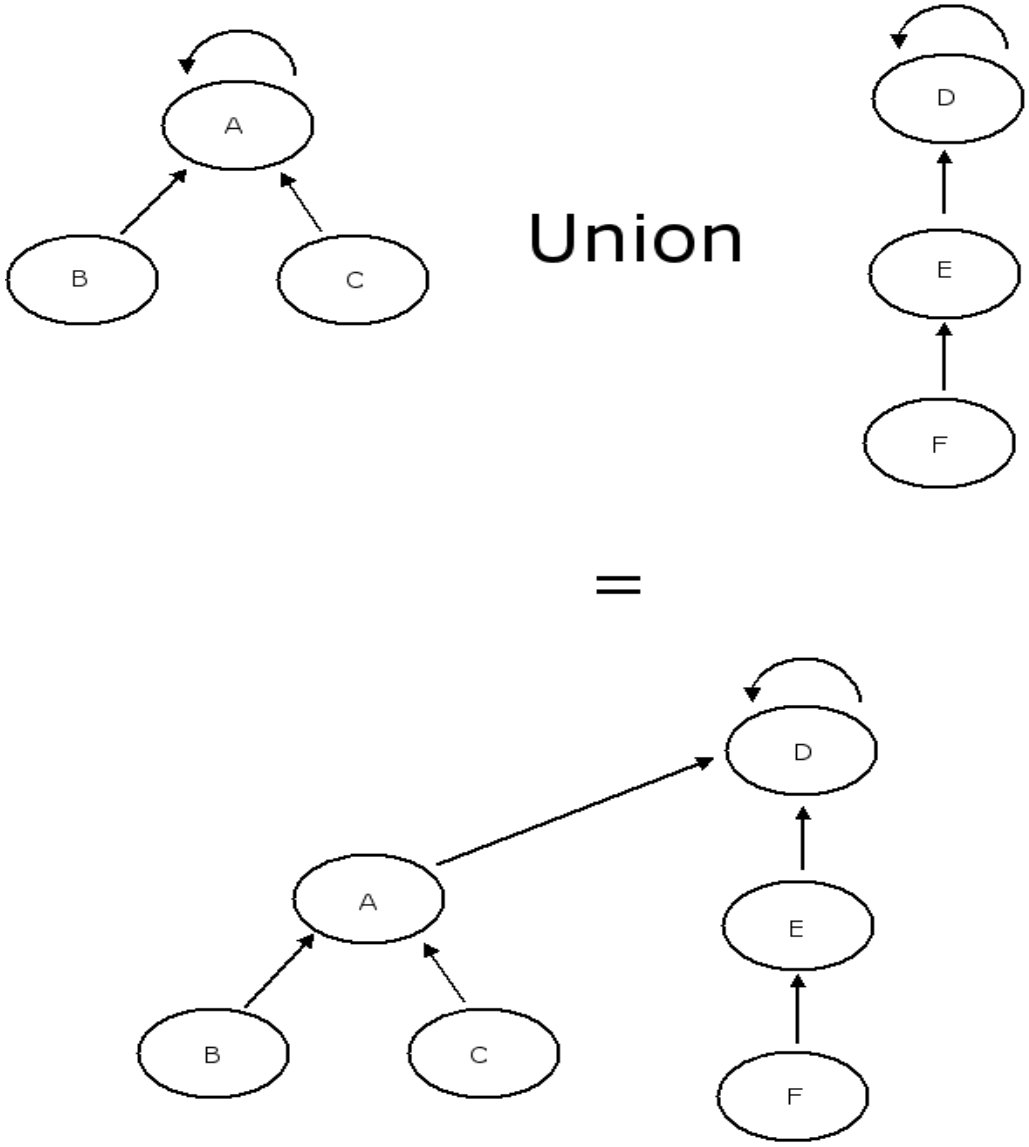


Figure 2.8: Two disjoint sets are unioned. The new representative is either A or D, in this case D is selected.

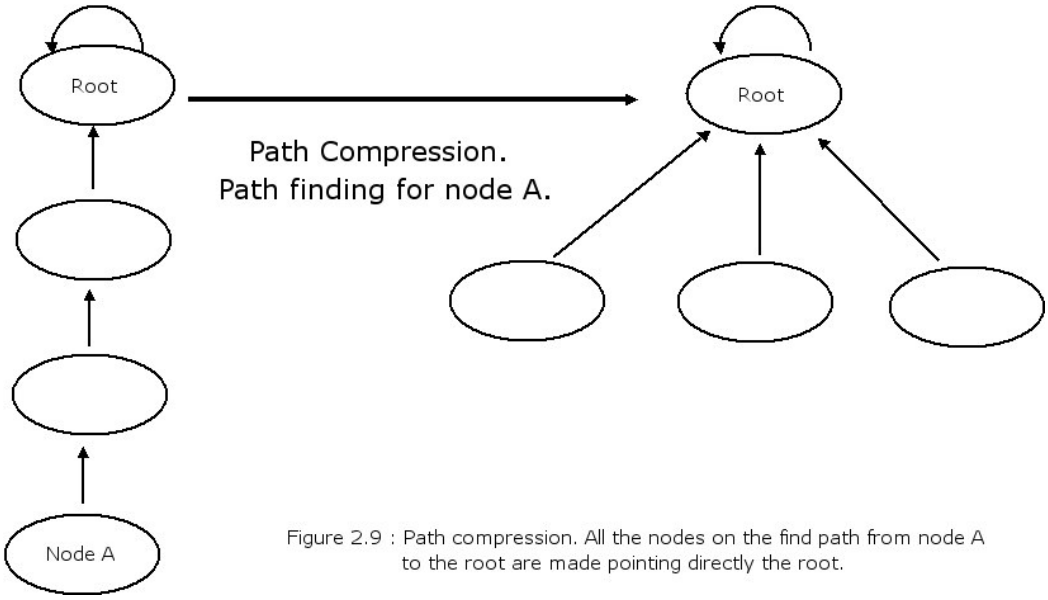


Figure 2.9 : Path compression. All the nodes on the find path from node A to the root are made pointing directly the root.

Each node maintains a rank which is an upper bound on the height of the node. When two sets are merged, the root with smaller rank is made to point to the root with larger rank, this technique is known as union by rank. For example in figure 2.8, node A's rank is two and node D's rank is one. node D is the root of the union set while node A points to node D. Union by rank minimises the algorithm running time [15].

The running time of creating a set is O(1), finding a root is O(logn) and unioning two sets is O(logn), where n is the total number of nodes [29].

In the disjoint sets blob algorithm:

segment = node
blob = set

The disjoint set blob algorithm uses the disjoint sets theories to merge overlap segments. Similarly as the old algorithm, the new algorithm scans segments from the top to bottom row, left to right each row. During the scan the algorithm compares each segment with all the segments on the previous row and merge them if they overlap. The overlap conditions are same as the old algorithm.

The differences come when the algorithm attempts to merge segments. Lets follow an example to see how the disjoint sets

algorithm merges segments for the first overlap case section. Refer to figure 2.4. A set with only segment A is created by the make set operation, the root is segment A and it points to itself, lets call this set as S(A). This set is merged (union) with S(BC) which contains segment B and C. The union operation needs to know the root of S(A) and S(BC). The root of S(A) and S(BC) can be found by the find set representative operation - given segment A and B. Path compression applies. Lets assume the roots are segment A and B. The union operation compares the rank of segment A and B. Since segment A has a rank of one while segment B has a rank of two, segment A is made to point to segment B (union by rank). Segment B is the root of S(ABC). See figure 2.10.

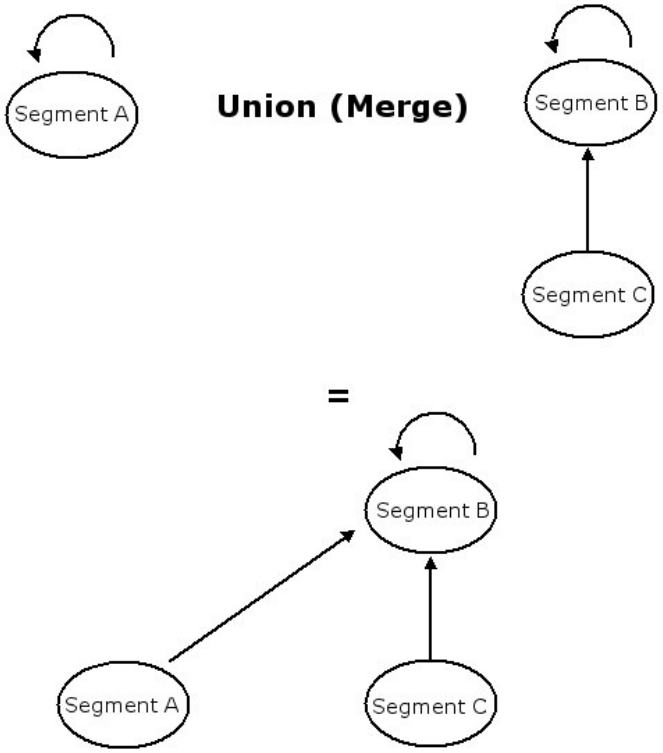


Figure 2.10: First overlap case merging by the disjoint sets blob algorithm. Correspond to figure 2.4.

Similar techniques apply to the second overlap case. Refer to figure 2,11. Since segment A belongs to set AC, the disjoint sets blob algorithm unions set AC with S(BDEFGHI). The root of S(AC) and S(BDEFGHI) are found by the find set representative operation — given segment A and B. Path compression applies. Lets assume the roots are segment A and segment F. The union operation compares the rank of segment A and segment F. Since segment F has a higher rank,

segment A is made to point to segment F (union by rank). Segment F is the root of the new union set. See figure 2.11.

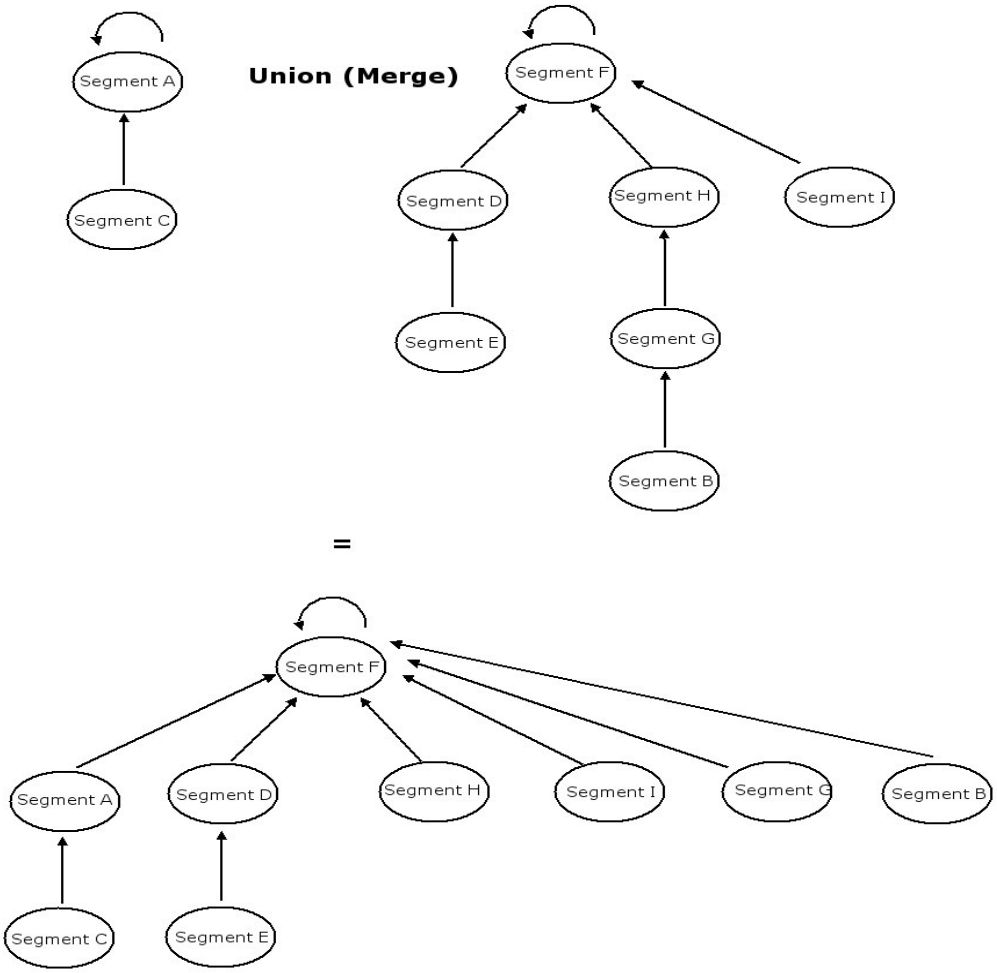


Figure 2.11: Second overlap case merging by the disjoint sets algorithm. Correspond to figure 2.5. Path compression applies to the find path segment B -> segment G -> segment H -> segment F.

Basically the operations required for the first and second overlap cases are identical. Disjoint sets blob algorithm is fast because the running time for the second overlap case is significantly improved. Only the roots need to be found, once they are found, one of them is made to point to the other. Finding the root is a fast operation because the height of the tree is usually small due to the path compression heuristic. In figure 2.11, only 4 comparisons are required by the disjoint sets algorithm (find-set operations), in contrast to 5 comparisons are required by the old blob algorithm. In the new blob algorithm, only the root is important and give information about the set.

The running time of the disjoint set algorithm is O(mlogn),

where m is the total number of make-set, union and find-set operations and n is the total number of make-set operations. n is equal to the total number of segments, in the worse case it is hw where h is the height of the cplane and w is the width of the cplane [29].

```
/*
   Copyright 2004 The University of New South Wales (UNSW) and
   National ICT Australia (NICTA). This file is part of the 2004
   team rUNSWift RoboCup entry. You may redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of
   the License, or (at your option) any later version as modified
   below.  As the original licensors, we add the followin
   conditions to that license: In paragraph 2.b), the phrase
   "distribute or publish" should be interpreted to include entry
   into a competition, and hence the source of any derived work
   entered into a competition must be made available to all parties
   involved in that competition under the terms of this license.
   In addition, if the authors of a derived work publish any
   conference proceedings, journal articles or other academic
   papers describing that derived work, then appropriate academic
   citations to the original work must be included in that
   publication. This rUNSWift source is distributed in the hope
   that it will be useful, but WITHOUT ANY WARRANTY; without even
   the implied warranty of MERCHANTABILITY or FITNESS FOR A
   PARTICULAR PURPOSE.  See the GNU General Public License for more
   details. You should have received a copy of the GNU General
   Public License along with this source code; if not, write to the
   Free Software Foundation, Inc., 59 Temple Place, Suite 330,
   Boston, MA  02111-1307  USA
*/
```

```
      // Merge two sets — set x and set y.
      inline void Merge(RunLengthInfo *x, RunLengthInfo *y) {
            Link(Find_Set(x), Find_Set(y));
      }

      // Link set x and set y based on union by rank.
      inline void Link(RunLengthInfo *x, RunLengthInfo *y) {
            if (x->rank > y->rank) {
                  y->nextSeg = x;
            }
            else {
                  x->nextSeg = y;
                  if (x->rank == y->rank)
                        y->rank++;
            }

      }

      // Find the root given a node in the set. Path compression
      // applies.
      RunLengthInfo *Find_Set(RunLengthInfo *node) {
            RunLengthInfo *root = cs;
            while (root != root->nextSeg){
                  root = root->nextSeg;
            }
```

```
            while (cs != root){
                  RunLengthInfo *temp = cs->nextSeg;
                  cs ->nextSeg = root;
                  cs = temp;
            }

            return root;
      }
```

Figure 2.12: Code fragment of : union (merge), link (union by rank) And find-set (path compression apply).

```
void jointSegments() {

      ......

      // Get the root of the current and previous segment.
      //    cs is the current segment.
      //    ps is the previous segment.
      RunLengthInfo *csRoot = Find_Set(cs);
      RunLengthInfo *psRoot = Find_Set(ps);

      // Overlap occurs if the roots have same color and the
      // segments are connected. If the roots are identical then
      // they have at least one pixel overlap.
      if (csRoot->color == psRoot->color && csRoot != psRoot &&
          ps->overlaps(cs)) {

            // If the current segment is not joined previously.
            // First overlap case.
            if (csRoot->blob_number == -1) {

                  // Copy the blob number from the previous segment.
                  csRoot->blob_number = psRoot->blob_number;

                  // Merge both segments in a single disjoint set.
                  Link(csRoot, psRoot);

                  // Update information.
                  blobinfo[psRoot->color][psRoot->blob_number].
                   Update
                  (cs->rawColor, cs->startIndex, cs->endIndex, row,
                   cs->length, cs->xsum, cs->ysum);

            // Second overlap case.
            } else {

                  // Merge the disjoint sets.
                  Link(csRoot, psRoot);

                  // Find the root of the new disjoint set.
                  RunLengthInfo *root = Find_Set(ps);

                  // This new root is assigned the blob number from
                  // the previous segment.
                  root->blob_number = psRoot->blob_number;
            }
      }

      ......

}
```

Figure 2.13: Simplified code fragment of the disjoint sets blob algorithm.

2.4: Performance

Three images were taken for comparison between the old and new blob algorithm. The first image has few noise and few segments. The second image has some noises and more segments. The third image is extremely noisy and over hundreds of segments. Clearly the first image requires the least processing power, the second image requires more, the third image requires the most. See figure 2.14, 2.15 and 2.16. Their running times are reported in figure 2.17, 2.18 and 2.19. This experiment was tested with the offvision software [1]. Since the offvision is not totally reliable, the experiment was done five times.

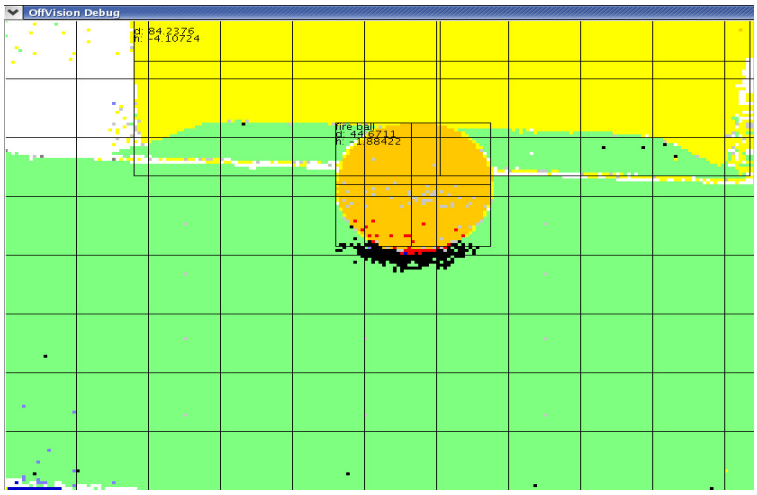


Figure 2.14: A simple camera image.

1Execute the vision module with a PC. Refer to [10] for more details.

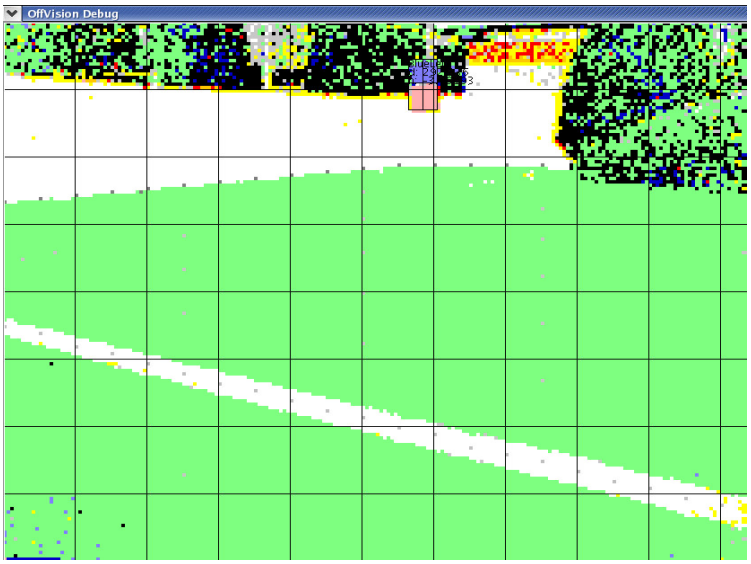


Figure 2.15: A camera image with some noises.

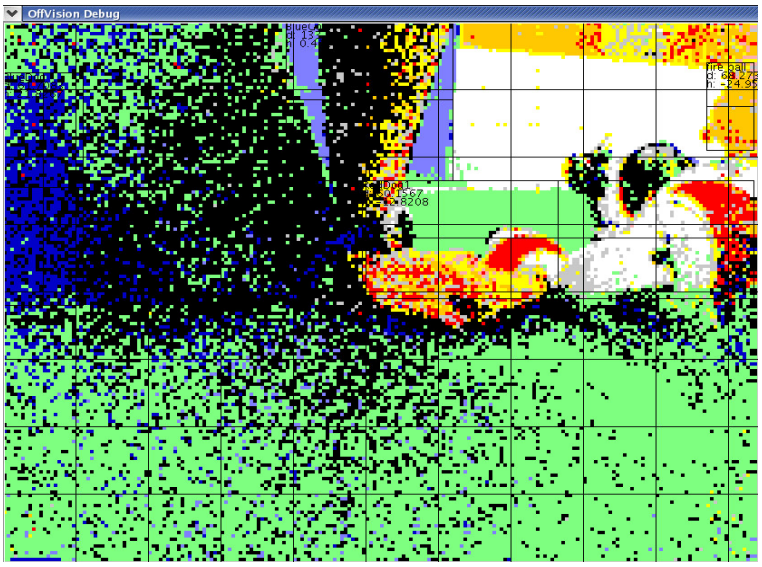


Figure 2.16: A complicated and noisy camera image.

						Average
Old algorithm	436	447	430	437	442	438.4
New algorithm	438	436	426	428	431	431.8

1.51 (2dp) % improvement

Figure 2.17: Blob algorithm running time in ms of figure 2.14.

						Average
Old algorithm	781	730	778	788	775	770.4
New algorithm	606	650	699	599	595	629.8

18.25 (2dp) % improvement

Figure 2.18: Blob algorithm running time in ms of figure 2.15.

						Average
Old algorithm	2741	2752	2747	2754	2741	2747
New algorithm	1467	1470	1499	1469	1589	1498.8

45.44 (2dp) % improvement

Figure 2.19: Blob algorithm running time in ms of figure 2.16.

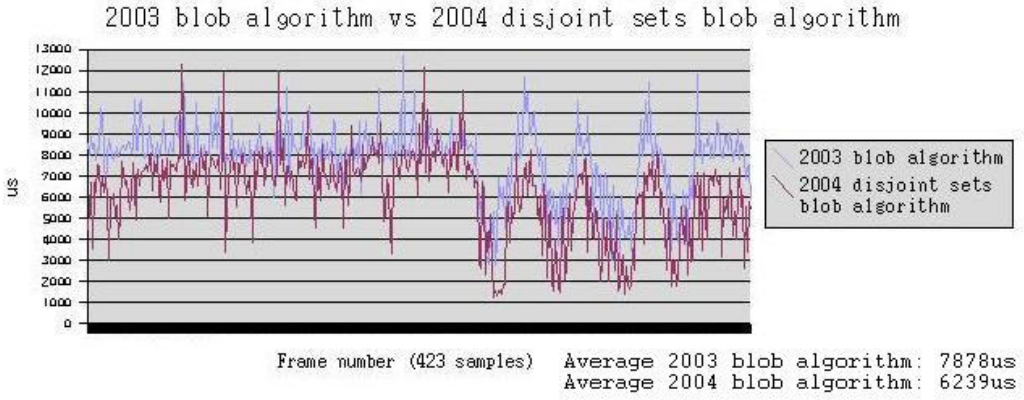


Figure 2.20: Running time of the old and new blob algorithm.

Clearly the disjoint sets blob algorithm runs faster than the

old blob algorithm. The actual improvement is depend on the quality of the camera image — good quality, little improvement and poor quality, significant improvement. Complex images contain more second overlap cases than simple images, hence the disjoint sets improvement becomes more significant (45.44% for figure 2.16). In figure 2.14 only few second overlap cases occur, since the old and new algorithm have identical running time for the first overlap case, not much improvements are made (1.51%).

2.5: Eight-side merging

Sometimes few pixels miss from the distortion. Since the blob algorithm only merge pixels vertically/horizontally, an object with few missing pixels may not be recognized.

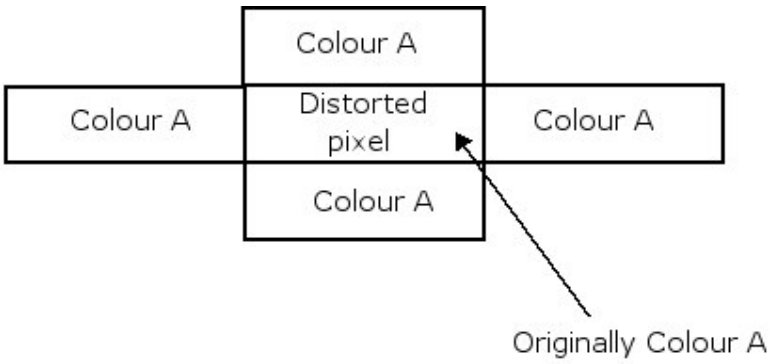


Figure 2.21 : A distored pixel causes the object not recognised.

Merge pixels vertically and horizontally is known as four-side merging. From observations sometimes the object may still be recognized if pixels are merged diagonally, such as in figure 2.20. Some experiments were tried with eight-side merging. Eight-side merging merges pixels vertically, horizontally and diagonally, only the overlap condition need to be changed.

However eight-sided merging is not used in the competition. It is only useful for the robot recognition [5] so the algorithm should be able to differentiate between colors. Due to limited time it is not implemented.

2.6: Future development

Eight-side merging with color differentiating may be tried in the future. It should improve the robot recognition which is currently not very good. The object recognition may be benefit from blobbing certain colors eight-way and others four-way, it was tried unsuccessfully this year, for more details refer to Jing Xu's thesis [8].

Chapter 3
Locomotion and Wireless

Section 3.1 to 3.3 present an overview of the previous and current locomotion module. I did not write any of the locomotion codes, but I need to understand the concepts in order to integrate them with the behaviors.

3.1: Locomotion overview

Locomotion module receives action instructions from the behavior module, buffer it and forward the instructions to the hardware through OPEN-R.

There are a number of walk types that a robot can have, the difference is the synchronization between the legs. For example a robot can walk by moving one leg at a time known as crawl walk [17]. From experiments, a robot walks fastest when its diagonally opposite legs lift up while the other pair must stay on the ground.

The robot walks by defining locus for each feet to trace out. The robot's speed is depend on the shape of its walk locus.

A step is finished when a feet completes one cycle of locus. The feet can only change direction at the end of a half step.

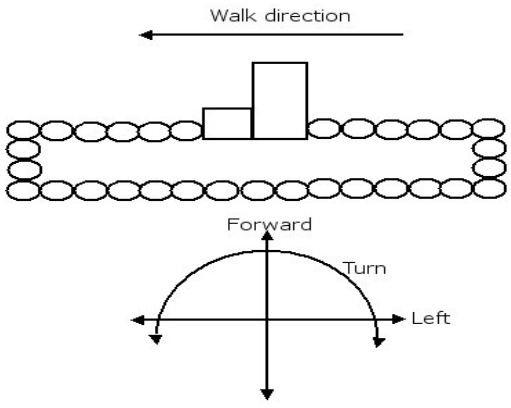


Figure 3.1: Walk locus. The feet traces the points on the walk locus. The three walk components are shown.

The robot has three walking components: forward, left and turn. The forward component allows the robots to move forward and backward. The left component allows the robots to move sideway. The turn component allows the robots to turn on the spot. The robot's movement is the combination of forward, left and turn component. Most of the walk types allow the robots to move with these three components. See figure 3.1.

3.2: ERS-210 locomotion overview

rUNSWift had invented four walk types for the ERS-210 robots. Each walk type defines an unique locus. They are:

3.2.1: Rectangular and Canter walk

Rectangular walk allows the robot to move in any direction but it is slow. Canter walk is similar to rectangular walk but the robot body moves in a sinusoidal wave pattern (page 233 of [13]). Rectangular and canter walk type defines a rectangular locus.

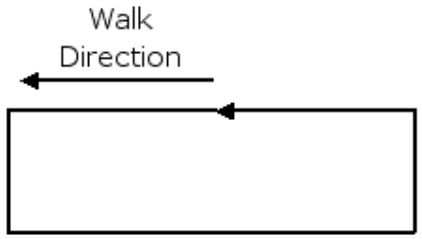


Figure 3.2: Rectangular and canter walk locus.

3.2.2: Zoidal walk

Zoidal walk defines a trapezoidal walk locus. This walk type allows the robot to move forward quickly, but it cannot turn very well.

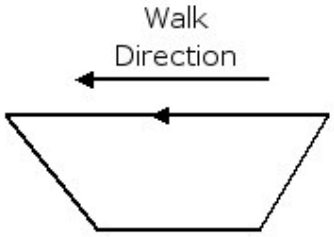


Figure 3.3: Zoidal walk locus.

3.2.2: Offset walk

Offset walk is the fastest walk type. It allows the robot to move forward at its maximum speed. It is a calibrated version of the zoidal walk. The locus for the front and hind legs are slightly different.

Interested readers are referred to [17] for more details.

Only canter walk and offset walk were used in the last year's strategies. Offset walk was fast, it was used for walking forward/backward and turning slowly. However, offset walk could not turn quickly, canter walk was used when the robot needed to turn quickly specially turning on the spot.

3.3: ERS-7 locomotion overview

None of the ERS-210 walk types could be used for the ERS-7 robots due to the physical differences. Instead better and faster walk types were invented. This year five walk types were used in the competition — normal walk, high gain normal walk, elliptical walk and high gain elliptical walk.

3.3.1: Normal walk

Normal walk is a slower walk type which moves approximately 15 cm per second. It is not useful for walking forward because it is too slow, this walk corresponds directly to the ERS-210's canter walk. This walk type is flexible as it allows the robot to move in any direction smoothly. Normal walk allows the robot to stop. Normal walk's locus is same as canter walk's locus.

High gain normal walk is not really a new walk type, it is just the normal walk with high gain. The high gain version produces a much faster walk than the low gain version.

3.3.2: Elliptical walk

Elliptical walk is a new walk type, its locus is a semi-circle. It is a fastest walk type, much faster than the normal

walk. Its maximum forward speed is 34cm per second. Elliptical walk's properties are similar as the offset walk — cannot turn quickly.

Although elliptical walk is fast it is very inflexible. It doesn't allow the robot to move sideway and control its forward speed. The robot always move forward/backward with its maximum speed, so there is no way to stop the robot with elliptical walk.

High gain elliptical walk produces a much faster walk than the low gain version.

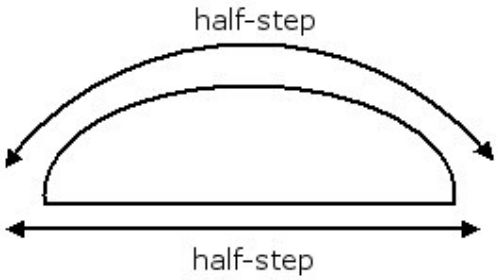


Figure 3.4: Elliptical walk locus.

For more details on the normal and elliptical walk, refer to [9] and [16].

3.3.3: Normal vs Elliptical walk

Both the normal and elliptical walk were used in the competition. Normal Walk is handful when the robots needed to make a large turn, move sideway or stop. Elliptical walk is useful when the robots want to walk forward quickly or make a small turn.

In a game the robot needs to switch the walk types dynamically. PID gains are also switched dynamically. If the battery current is high, all the leg motors are switched into a smaller set of PID gains to avoid battery overcurrent. When the battery current is low, the gains are switched to high gain for stronger and faster motion. See the section on dynamic gain (section 5.7) for more details.

3.4: Wireless

3.4.1: Robot communication

Previously TCP[1] was used for robot wireless communication. Earlier of the year rUNSWift had a serious wireless problem with the latest Sony TCP remote processing. For some reasons three or more robots could not be communicated. The communication between only two robots was fine.

In the Australian Open, UTS allowed the rUNSWift to run remote processing with an older version of OPEN-R. With the older OPEN-R, the robots were able to connect to each other. The Newcastle team [18] explained this problem to the rUNSWift team. The problem caused by changes in remote processing in the latest OPEN-R, the latest OPEN-R doesn't allow the robots to broadcast packets to each other. One solution is to disable packet broadcasting, so instead of robot A broadcasting a packet to robot B, robot C and robot D, robot A would send the packet only to robot B, and then robot B forward the packet to robot C, and finally robot C forward the packet to robot D Although this method would work it is not recommended, since it is possible that Sony will change the remote processing method again, then future teams must re-implement the wireless protocol. A better solution is to switch from TCP to UDP[2], so that the robots do not need to use the Sony TCP remote processing.

After the Australian Open, the team switched the protocol from TCP to UDP. UDP allows the robots to broadcast packets. The only drawback of UDP is the fact that it doesn't support packet loss recovery [19]. Fortunately this should not be a problem unless the packet loss rate is significant. Any loss packet would be replaced by the next received packet.

3.4.2: Wireless speed

The 2003 wireless speed was 500ms delay (page 165 of [13].

1Transmission Control Protocol. A reliable network protocol for data transfering.

2User Datagram Protocol. A network protocol for data transferring .

This year the speed rises to approx 10 to 100ms delay. Faster wireless transmission allows the behavior module to place more emphasis on wireless communication, in theory should lead to better team cooperation.

However due to network congestion, the wireless speed in the world open was actually worse than the previous year. This sudden change had impacts on the robot's behavior. For example the team cooperation was poor (section 5.13.4).

3.4.2: New gamecontroller

This year the gamecontroller has modified slightly. The new gamecontroller allows the operator to change the team color and change the state to the ready state. Refer to page 282 of [10] for more details.

Figure 3.5: Gamecontroller 7.2.

3.5: Kicks

3.5.1: Introduction

Kicks can be divided into two categories. The first category includes all kicks that are simply playback of joint angles. The second category includes all kick that involve decision making.

First category (playback of joint angles)

- Lightning kick
- Front kick
- Chest push
- Dive kick
- Hand kick
- Side kick
- Head kick

Second category (kicks involve decision making)

- Dribble
- Turn kick
- Visual opponent avoidance kick

The following sections describe the first category kicks. The second category kicks are described in the next two chapters.

3.5.2: Lightning and front kick

Lightning and front kick were invented in the year 2002. These kicks are very powerful for the ERS-210 robots.

Lightning kick consists of two steps. In the first step, the robot raises its front paws, bent its knees and looking down on the ball to trap it under the robot's head. In the second step, the robot raises its head and bring the front paws down to shoot the ball forward. The rear legs are brought inward. See figure 3.6. Lightning kick is a very fast kick.

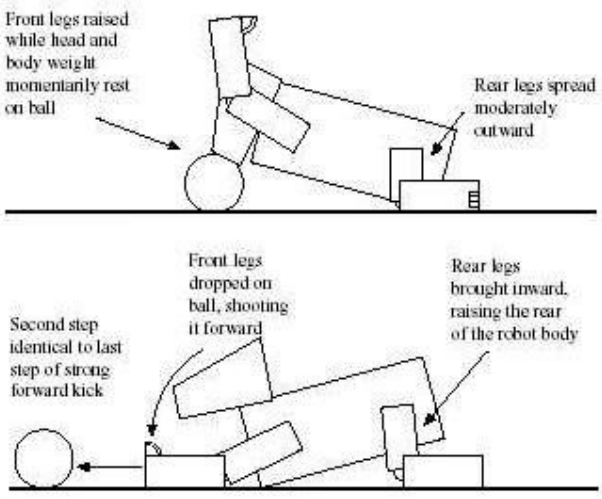


Figure 3.6: Lighting kick. (Image courtesy of 2002 UNSW RoboCup report [10])

Front kick is similar as the lightning kick. Firstly the robot lay down on the ground, reaching its front paws forward and spreading its rear legs outward from the body. This step allows the robot to trap the ball with the front legs. After the ball is held, the robot raises its front paws and lowering its head to trap the ball. After this step is completed, the robot brings its front paws down and shoot the ball forward. See figure 3.7.

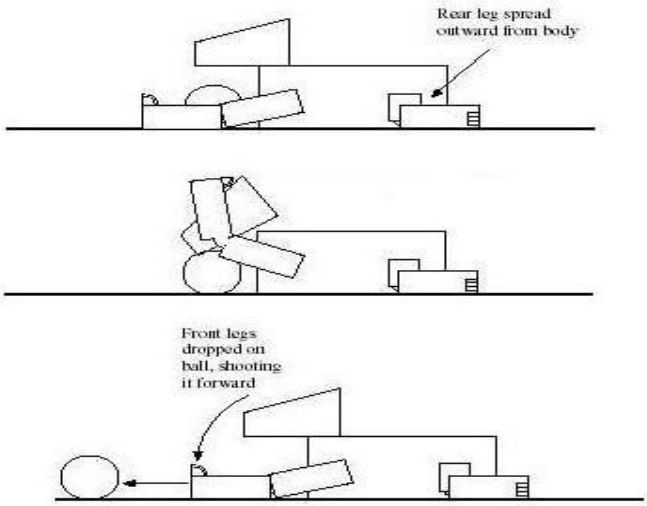


Figure 3.7: Front kick. (Image courtesy of 2002 UNSW RoboCup report [10])

Front kick requires that the ball is held first, otherwise it may not execute correctly. Lightning kick requires that the ball is held with the head by approaching it while ball tracking. Refer to the 2002 UNSW RoboCup report for more details (page 100 of [10]).

Unfortunately lightning and front kick are not useful for the ERS-7 robots. These kicks need a reliable ball grabbing which cannot be achieved with the ERS-7 robots. See the next chapter for more details.

3.5.3: Chest push

Chest push was invented in the year 2001. To begin, the robot moves its body back, storing momentum while the paws stay in place. When this step is completed, the robot quickly push its body forward, move the front paws slightly backward and hit the ball forward.

Chest push is relatively fast, but its power is limited. It cannot hit the ball too far away. Due to the limited range, in the year 2003 it was not used for shooting, but used for short passes to teammate (page 239 of [13]).

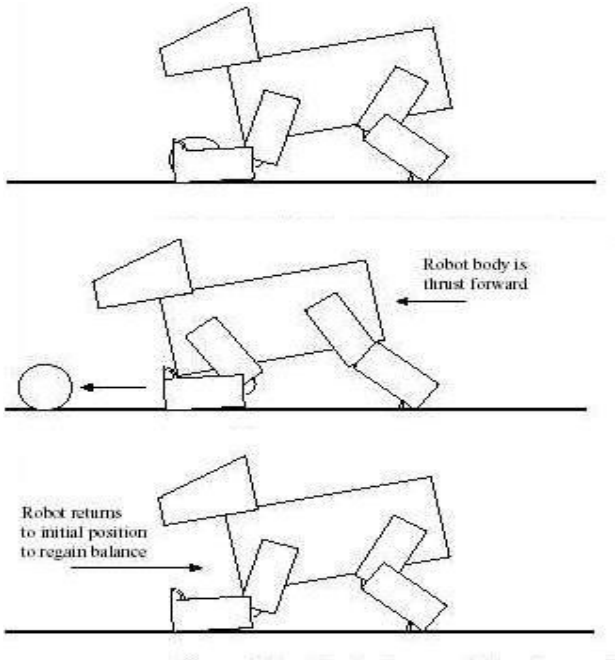


Figure 3.8: Chest push. (Image courtesy of 2002 UNSW RoboCup report [10])

3.5.4: Dive kick

Dive kick is invented by Pham.K.C after the Australian Open for his RoboCup open challenge [9]. This kick is not new however, the 2002 UNSW RoboCup report has mentioned it (page 105 of [10]).

Dive kick allows the robot diving forward, kicking the ball by head. To begin, the robot moves its front paws forward, sliding the paws on the ground. The head moves forward also. The back paws are bent slightly, supporting the body. See figure 3.9. After this step is completed, the robot returns to its initial position by diving backward.

This kick hits the ball forward. Its power is excellent since the robot's body weight is used. It is relatively fast. This year, dive kick has replaced the lightning and front kick for kicking the ball forward. Interested readers are refer to [6] and [9].

Figure 3.9: Dive kick.

3.5.5: Hand kick

There are two versions of hand kick: left and right hand kick. Left hand kick kicks the ball approx 30 degrees to the right. Right hand kick kicks the ball approx 30 degrees to the left.

Before hand kick can be executed, the ball must be grabbed. After the ball is held, the robot drops down on the ground, the front paws are reached forward and the back legs are bent to support the robot's weight. If the robot wants to execute left hand kick, it uses its right front paw to kick the ball to left hand side, while raising the left front paw. When the ball is under the left front paw, the robot quickly bring it down and hit the ball approx 30 degrees to the right hand side. See figure 3.10. Similarly for the right hand kick. Hand kick is only useful when the ball is grabbed. Hand kick and sideway kick (next section) are the only two kicks that are used after the robot has gained possession of the ball.

Currently handkick is used after the ball grabbing and turning. Sometimes the robot wants to turn the ball around, it runs toward the ball, grab it, turn it and then perform a handkick to hit the ball toward the target side. The paw the robot uses to hit the ball is depend on the turning direction. If the robot turns clockwise, it would execute a left hand kick. If the robot turns anticlockwise, it would execute a right hand kick.

Hand kick is invented by Pham.K.C after the Australian Open for his RoboCup open challenge [9].

Figure 3.10: Hand kick. 1. Drop on the ground. 2. Hit the ball sideway 3. Shoot the ball.

3.5.6: Sideway kick

In the 2003 RoboCup final University of Pennsylvania (UPenn) vs rUNSWift, the UPenn team used a very powerful sideway kick [28]. Their robots were able to kick the ball past the rUNSWift robots. Since this kick is useful, rUNSWift has adopted it this year.

Sideway kick has two versions: left and right sideway kick. Left sideway kick allows the robots to hit the ball to the right hand side. Right sideway kick is similar, it allows the robots to

hit the ball to the left hand side.

Sideway kick is a very simple kick. If the robot wants to perform a right sideway kick, it bends its right front paws while bending and moving its rear legs slightly backward, storing momentum. After this step is completed, the robot hits the ball with its right front paw from the right hand side, hence hitting the ball toward the left hand side. The left paw is moved slightly backward and inward, supporting the robot's body. See figure 3.11. Left sideway kick is similar.

Sideway kick is fast and powerful. The robots do not need to lay down on the ground, so the robots can recover very quickly. It is one of the major kick used by rUNSWift currently.

Currently sideway kick is used after the ball grabbing and turning. Sometimes the robot wants to turn the ball around, it runs toward the ball, grab it, turn it and then perform a sideway to hit the ball toward the target side. The paw the robot uses to hit the ball is depend on the turning direction. If the robot turns clockwise, it executes a left sideway kick. If the robot turns anticlockwise, it executes a right sideway kick.

Sideway kick is ported by Chan.K.C, see [6] for more details.

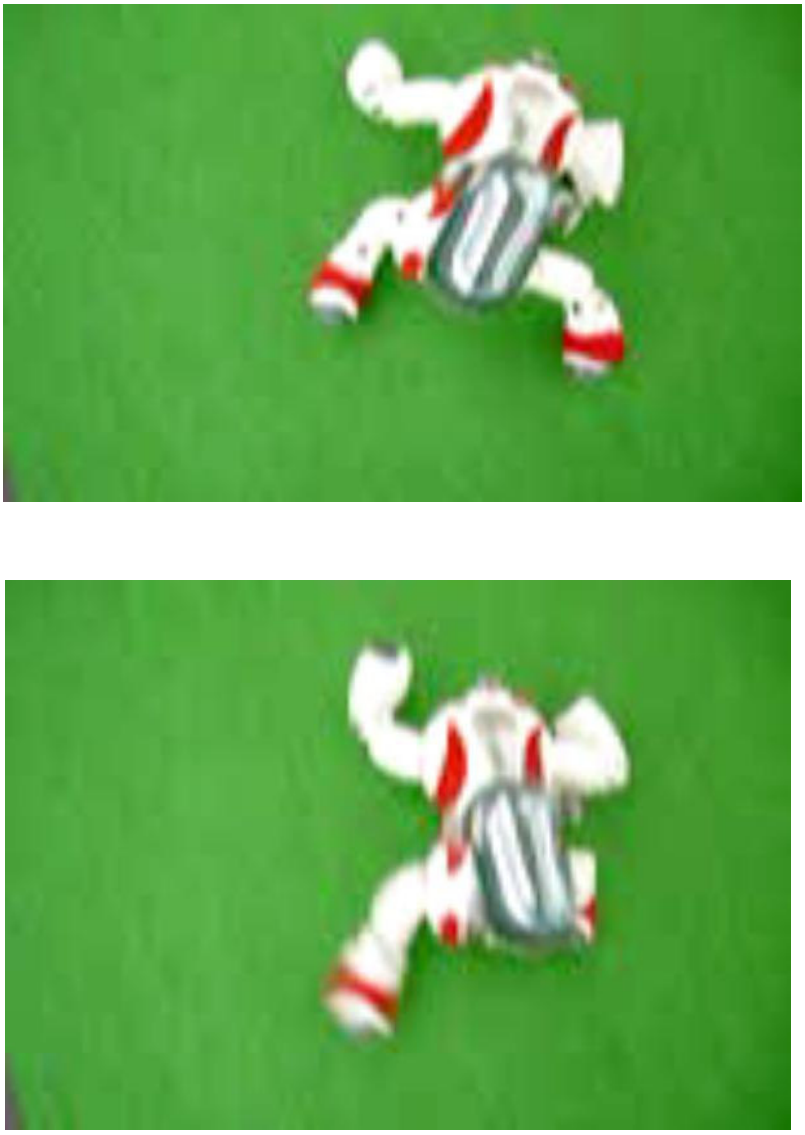


Figure 3.11: Right sideway kick. Left sideway kick is similar, but the paws are changed.

3.5.7: Head kick

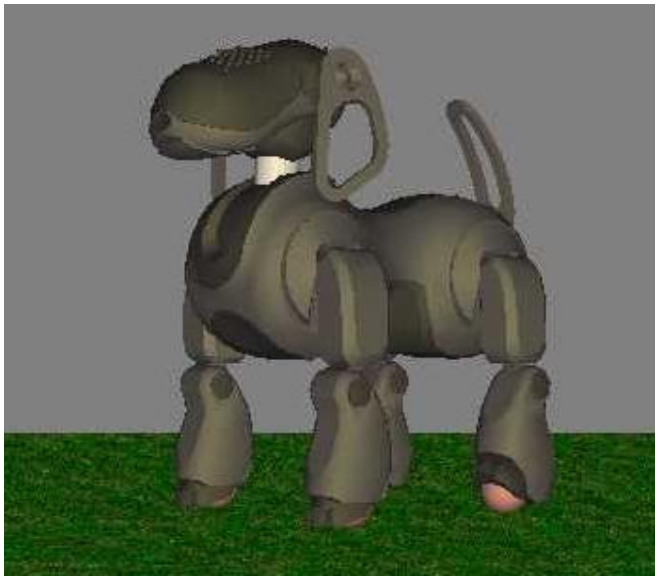
Head kick is a new kick developed for rUNSWift robots. There are two versions of this kick: left head kick and right head kick. The robots hits the ball with its head. The left head kick hits the ball toward the left hand side relatively straight. The right head kick hits the ball toward the right hand side relatively straight.

In the event of left head kick, the robot hits the ball from the right hand side of the ball and hence hit the ball toward the

left hand side. Right head kick is similar.

Since the ERS-7 robot has a larger head, longer neck and an extra degree of head movement freedom, head kick is a very powerful kick for the ERS-7 robots. This kick would be very useful when the robot faces the left or right edge, if the robot faces the left edge, right head kick would quickly hit the ball toward the target side. Similarly for the right edge. Head kick is not just powerful, it is also very fast, since the robot does not need to lie down on the ground or grab the ball. This year's world champion Germany team had used this kick very successfully in this year's competition.

This kick was developed during the competition, and hence not enough time to put it into the strategies. The future teams should integrate it into the strategies.



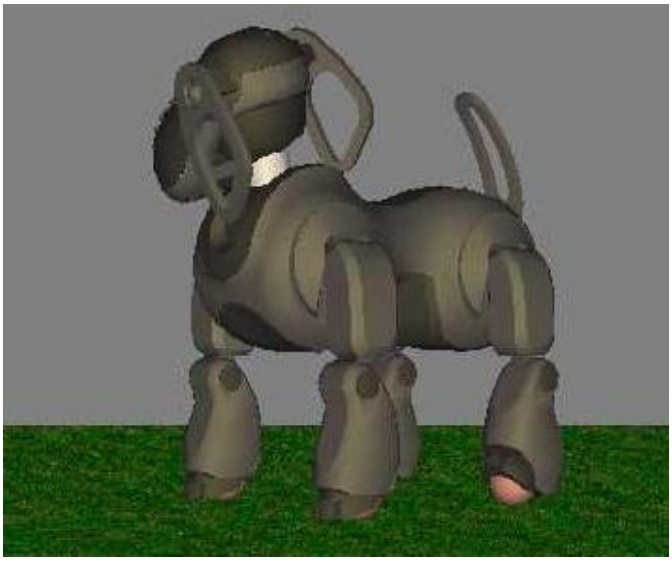


Figure 3.12: Left head kick. The robot moves its head to the right hand side and then quickly pan its head toward the left hand side and hence hit the ball toward the left side.

Chapter 4
Behavior before Australian Open

The RoboCup Australian Open is an annual event where the Australian universities compete against one another [20]. In 2004, it was took place on April 16, 2004. The behavior had underwent significant changes after the Australian Open. Since the behavior before and after the Australian Open is completely different, they are separated into two chapters. This chapter describes the behaviors before the 2004 Australian Open. The next chapter describes the behaviors after the 2004 Australian Open.

4.1: Behavior summary

The rUNSWift behaviors are specified in a large decision tree. Decision tree is a hierarchy of nodes, each node makes a decision which is based on the robot's localisation, wireless, states, roles, and strategies. The overall team strategy is the combination of each robot's decision tree. See section 5.3 for more details.

Due to various difficulties encountered, the decision tree and hence the strategies used before and in the Australian Open was based on the previous year strategies. In fact rUNSWift used the same behavior code as the previous year, with slight modifications. The difficulties encountered:

- Battery overcurrent

  rUNSWift encountered frequent hardware crashes due to the fact that the battery current was too high. The robots crashed too quickly for any serious behavior development. See section 4.2 for more details.

- Unstable walk

  rUNSWift could not developed a stable walk until after the Australian Open. Everytime when a significant change occurred

to the locomotion module, the behavior must also changed accordingly. Only the normal walk was able to develop and integrate successfully before the Australian Open. However normal walk is slow. Elliptical walk which is a much faster walk, was invented two days before the Australian Open and hence there was not enough time to integrate this walk in the behavior. Interested readers are referred to [9] and [16] for more details.

- Poor vision
    - The robot could not differentiate between the red and orange color. It recognized the red robots as orange ball and charged toward the red robots [8].
    - The fact that the vision module could not distinguish between black and blue implied the blue robots couldn't be recognized [5][8].
    - The 2003 sanity checks were broken, fake objects were recognized [5].
    - The ERS7 camera images are highly vulnerable to noise and distortion near the edges of camera images. Hence the robot could not recognize the beacons and other objects very well. For example pink blobs of beacons were seen as orange, hence recognized as the orange ball [8].
- Localisation problem

    This year the two mid-field beacons have been removed, also the height of the walls surround the field are halved. It has an significant impact on the robot localisation because the robots have less information and the camera images would be more noisy. Robots frequently mislocalised near the corners. Although the field edge detection was invented, it was implemented successfully only after the Australian Open [7]. Very difficult to test a new strategy when the localisation was completely mislead.

Behavior module relies heavily on the low levels, if they are not reliable then not much behavior improvement can be done.

Unfortunately the previous year behavior codes could not be

used directly since there are physical differences between the ERS-210 and ERS-7.

Due to the physical differences and other impacts, some of the previous skills were not performing effectively this year. Those skills were rewritten, replaced or even removed:

- Since the ERS-7 robots cannot grab the ball with their front paws, they must use their head to grab. See section 4.3 for more details.

- Lightning kick, front kick and turn kick were removed because the ERS-7 robots cannot grab the ball with their paws. They were replaced by chest push and paw kick. Chest push was executed where previously the robot would performed lightning kick. Few days before the Australian Open, more paw kicks were put in to filter out the chances of any forward and lightning kicks. Interested readers are referred to Chan.K.C's thesis report for more details [6].

- Hover to ball was rewritten. The new hover to ball minimised the battery current. Refer to section 5.8 for more details.

- Dribble was rewritten since the previous dribble could not grab and turn with the ball. Dribble is a skill allowing the robots to push the ball toward the target goal. Refer to [6] for more details.

The fact that the two middle beacons were removed also had an impact on the behavior. The active localise was revised and reworked, eliminating the decisions involved the two middle beacons. As a consequence the robots would never look for the two middle beacons.

4.2: Battery overcurrent

4.2.1: Overview

Earlier of the year rUNSWift had a very serious robot hardware crash. The robots crashed so easily and so quickly that it was virtually impossible to play a game. rUNSWift was not able to play a single full game before the Australian Open. Essentially no robot could survive for a full game. Most of the development time before

the Australian Open was concentrate on only making the robots to survive for the whole game.

4.2.2: Detail description

The robot's motors operate by drawing current from a battery. The level of the current is determined by the hardware's requirements. When the motors are operating quickly (ie. fast motions), more current must be drawn. When the motors are operating slowly (ie. slow motion), less current is required.

Battery overcurrent occurs when the OPEN-R detects that the battery current is over a threshold. If this occur the hardware automatically shut down the robot, protecting it for any possible damage. The software has no control of the shut down. Since OPEN-R is not open source, rUNSWift was not able to determine the threshold until Sony reported it is 3900ma on its web site [21]. Sony has released the battery overcurrent code fragment, see figure 4.1 [21].

```
static const int OVER_CURRENT_THRESH = 3900; // 3900mA for ERS-7
static const int OVER_CURRENT_THRESH = 3000; // 3000mA for ERS-210/220

OPowerStatus pstatus;
OStatus result = OPENR::GetPowerStatus(&pstatus);
if (abs(pstatus.current) >= OVER_CURRENT_THRESH) {
        OSYSLOG1((osyslogERROR, "BATTERY_OVER_CURRENT"));
        Shutdown();
}
```

Figure 4.1: Sony OPEN-R battery overcurrent code fragment. (Code fragment courtesy of Sony OPEN-R AIBO SDE homepage [21])

Not all the motors were responsible for the battery overflow. The robots only automatically shut down when it was moving. In another words only when the leg motors were moving. If the robots remained stationary, the battery current would never overflow. Head and other non-leg motors apparently had nothing to do with the battery overflow.

Battery overcurrent occurred frequently, rUNSWift did not see a single game without a robot crashed due to battery overcurrent. Possible reasons:

- Surface friction.

  rUNSWift's lab has a hard surface. When the robots walked on a hard surface, the motors experienced a strong friction, the motors must drew more current from the battery. This is believed to be the major reason of the battery overcurrent since the robots would never crashed if they were lifted up such that their legs had no contact with the ground (ie. no surface resistance, only air resistance). In the Australian Open, rUNSWift did not encountered any battery overcurrent, partly because UTS's lab has a soft surface.

- Locomotion implementation

  Some other teams reported that the battery overcurrent only occurred occasionally for their robots. The Germany AIBO team [23] stated that they did not experienced any battery overcurrent at all, in an email sent to the RoboCup mailing list soon after the 2004 RoboCup German Open [24]. From the videos downloaded at [23], the AIBO robots walked faster than the rUNSWift robots. Apparently other teams had developed better walks. Their robots could walked faster while drawing less battery current.

- OPEN-R designers

  Apparently the early OPEN-R designers expected the robots only needed to perform simple and slow motions. Programming the robot to play RoboCup was not something that the designers expected. In RoboCup, the robots must walk quickly and able to withstand the forces arised by crashing with different objects such as opponent robots.

- Heavy weight

  ERS-7s is much heavier than the old models. The leg motors must draw more current in order to move the robot. Some kicks such as lightning and front kick require lots of power when the robot gets up on the ground after finishing the move since the ERS-7 is just too heavy.

- Strong motors

  ERS-7 has very strong motors. Clearly stronger motors

require more battery current.

Although ERS-210 also crashes when the current overflow, rUNSWift did not encountered any problem. This is because the ERS-210s are light and their motors are not as strong as ERS-7's motors.

Although Sony had claimed the battery current overflow is the absolute of 3900ma, the team had seen the robots survived even though their battery current was less than -5000ma before the release of OPEN-R 1.1.5 r2. The team also had seen the robots crashed when their last recorded current was only slightly less than -1000ma. Most of the time the robots crashed when their current was less than -3000ma, rUNSWift did not see any crash when the current stayed above -1000ma. A possible explanation is the fact that the battery current overflow conditions are different for the OPEN_R 1.1.5 r1 and OPEN_R 1.1.5 r2.

Apparently the battery current accumulate over time, the robot's action may not have an immediate effect on the current level. Over time, if the robot continually perform the action, the battery current would change accordingly. For example, assume the robot has walked forward for few minutes, its current is approx -2500ma. Lets assume when a robot pauses, its current is approx -1000ma. The robot suddenly pauses, in theory the current should rise to -1000ma immediately. In fact the current would slowly rise to -1000ma , usually in 5 to 10 seconds, sometimes more and sometimes less. The battery current level is depend on the previous battery current level and previous actions. The robots need some time to cool down.

Battery current is extremely unpredictable. Different current readings were recorded for a same series of actions.

From practical observations, apparently the OPEN-R 1.5.1 r1 (not the OPEN-R 1.5.1 r2) updates the current reading every 45 or 90 vision frame, during the interval the current reading remain unchanged. See figure 4.2.

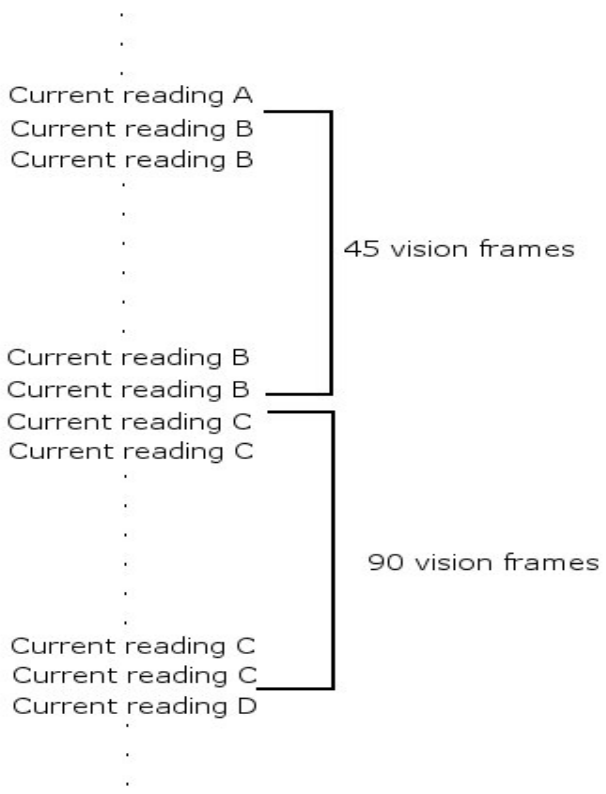


Figure 4.2: OPEN-R updates battery current reading every 45 or 90 vision frames.

4.2.3: Detecting battery overcurrent

Some experiments were tried to detect the battery current overflow before it occurred.

- Determine the overflow threshold

  Determining the overflow threshold was the very first thing rUNSWift did. This information could be useful since it could be used for the dynamical gain (see section 5.7). Unfortunately rUNSWift was not able to determine it successfully. Determining the overflow threshold was extremely challenging if not impossible. The current reading only updated every 45 or 90 vision frames, hence the current reading given by the OPEN-R might not be updated. For example if the robot performes three actions, action A, B and C. Assume the current update takes place after action C has completed, the current reading may be related to only action A, or action A and B or action A and B and C. Also when battery overflow occures, the robot immediately crashes

leaving no information.

The fact that the battery reading is not predictable also contributed the difficulties in determining the overflow threshold. Sometimes a series of actions might lead to current overflow, but sometimes it might not, even though the robot performed the actions under the same environment (eg. same carpet) and the battery readings were very close. An experiment was tried, after the robot had booted up, it repeatedly performed lightning kick. Sometimes the robot could only do one lightning kick and crashed. Sometimes it could survive for five or more lightning kicks. However the robot could not survive for more than ten lightning kicks. A guess might be related to how the current was building up. Sometimes it builds up quickly, and sometimes it builds up slowly. Details about how the current builds up and down is unknown.

From observations and experiments, rUNSWift estimated the overflow threshold was somewhere between -3000ma to -4000ma. The exact value according to Sony is ±3900ma [21].

- Assume overflow occurs when OPEN-R updates the current reading

From experiments, it was found that the hardware only shut down when OPEN-R updated the current reading which took place every 45 or 90 vision frames. An experiment was tried — assume overflows occurred every 45 and 90 vision frames and do something before it occurred. This method worked in terms of detecting the current overflow. However this approach did not help to avoid the battery overflow as will be explained in next section.

4.2.4: Preventing battery overcurrent

Some experiments were tried to prevent the battery overflow occurred, unfortunately none of the method worked perfectly. Before describing the experiments it would be a good idea to discuss how Sony reacted to the batter overflow. The first OPEN-R version designed for the ERS-7 robots is OPEN-R SDK ERS-7 Beta. On match, 2004, Sony released the OPEN-R SDK 1.1.5 r1 for ERS-7 and ERS-200 series. Both versions leaded to battery overcurrent. Due to the complaints from some of the RoboCup teams [25], Sony responsed it was possible to make the jam detection routine (ie. battery

overcurrent) *"less strict by creating the file "/OPEN-R/SYSTEM/CONF/VRCOMM.CFG" and writing a single line, "JamDetectionHighThreshold" in the file"* [25]. The rUNSWift team tried this suggestion, it certainly allowed the robots to survive longer than originally, for example a single robot was able to survive for the whole game with low gain normal walk. Unfortunately it was not enough to allow all four robots to play a whole game without crashing. Few days before the Australian Open, Sony released the OPEN-R SDK 1.1.5 r2. Sony claimed *"the high-current detection threshold in the OPEN-R SDK 1.1.5 r2 has been changed and ERS-7 is less likely to shut down.* [26]". This latest OPEN-R allowed the robots to play a whole game without crashing.

- Slowing down the robots

  This was done by setting a maximum speed. Before VRCOMM.CFG was suggested by Sony, from experiments, rUNSWift found that if the low gain normal walk was used and the maximum speed for each walk component (forward, left and turn) was four, the robot would not crashed unless it performed kicks or hitted objects such as the field border. Notice that the speed is in cm per locomotion step. After VRCOMM.CFG was used, the maximum forward speed could be set to seven, the maximum left and turn component could be set to ten.

  Walking slowly clearly helped to minimise the battery current. However this approach was not used in the Australian Open because the robots walked too slowly. The robots walked extremely slowly when its maximum speed was four, in particular with the low gain normal walk. Low gain normal walk is already a very slow walk type, rUNSWift could not afford to make the robots walk even slower. Walking slowly doesn't fit rUNSWift's strategy which emphasis on speed and aggressive.

  This approach could not prevent the battery overcurrent, only minimised the chances it would occurred. The robots still crashed if it performed high energy consumed kicks such as the lightning kick or hitted other robots. In fact before VRCOMM.CFG was introduced, even setting the maximum speed be 1 cm per step would still resulted in battery overcurrent if the robot crashed with other objects. Apparently when the robots were crashing into each other known as leg stuck, the leg motors experienced a large resistance and drew more current to

counter it.

Simply programming the robots to walk slowly is not a good idea. A better idea is to walk slowly only when the current is high, walk quickly when the current is low. This technique is known as dynamic gain.

- Pausing the robot

As mentioned earlier battery overflow only occurs every 45 or 90 vision frames. An experiment was tried — always paused the robot between 35^{th} - 45^{th} and 80^{th} - 90^{th} vision frames. The idea was to lower the battery current so that the current would be lower than the overflow threshold when OPEN-R was trying to determine whether it was necessary to shut down the robot.

This experiment was unsuccessful, it gave absolutely no improvement. Firstly the robot would paused frequently. Secondly rUNSWift found that if the current is high, nothing could prevent the robot to crash. Thirdly, pausing the robot in this fashion would not have any effect on the battery current. As mentioned earlier, the battery current is accumulative, it depends on the previous battery current and actions. The robots need at least few seconds to cool down. Since ERS-7 processes 30 vision frames per second, the 10 frames pauses between 35^{th} - 45^{th} translate to 0.1 second. Finally action instructions sent from the behavior to locomotion may not execute immediately, the actions may be buffer and execute later when the locomotion module is ready.

Pausing the robots when the battery current dropped below a threshold was also tried. This method did not work since rUNSWift was not able to determine the overflow threshold (section 4.2.3). Of course rUNSWift could estimated the threshold. Unfortunately estimating overflow threshold is tricky. If it is too low, then the robot would never paused. If it is too high, then the robot would paused too frequently. It is important to note that battery overflow only occur when the current is negative.

- Stuck detection

The robots crashed very easily if they hitted other objects such as the opponent robots and field border. Stuck

detection was developed to detect obstacles. See section 5.6 for more details. Stuck detection allowed the robots to detect and evade obstacles. It could not prevent crashes due to resistance from walking on the carpet.

- Low PID gain

  ERS-7 models have two servo gains, low and high gain. High gain gives much more powerful and fast motions but it consumes too much power. Low gain requires much less current than the high gain. But low gain also gives slower motions specially it is used for the normal walk. Before and in the in the Australian Open, low gain normal gain was the only choice except the match against UTS . High gain normal walk would crashed the robots too easily, even the VRCOMM.CFG file did not help. Unfortunately before the OPEN-R SDK 1.1.5 r2 was released, battery overcurrent could not be prevented with low gain motions.

- Dynamic gain

  A technique allowing the robots to dynamically switch between the servo gains was invented. Refer section 5.7 for more details.

  Although dynamical gain is useful, it is useless before the OPEN-R SDK 1.1.5 r2 was released. This is because before it was released even the most stable low gain normal walk would crashed.

- VRCOMM.CFG

  Although creating the VRCOMM.CFG file did not prevent the battery overcurrent, it allowed the robots to survive much longer than originally. Before it was released, the robots would crashed even if they were walking very slowly (less than 10cm per second). The file allowed the robots to walk with low gain normal walk's maximum speed (15cm per second). However this file is not sufficient to prevent crashes due to robot charging and high gain motions.

- Improving the locomotion module

  Parts of the locomotion module was redesigned. For

example:

- Develop an offline locomotion simulator.
- Trim the steps if the robot attempts to step too far.
- Perform binary search to find the longest reachable step size.

These improvements were made by my rUNSWift team members: Uther.W and Kim.C.K. Refer to Kim.C.K's project report for more details [9].

The improvements certainly helped. Before the improvements were made, battery current could dropped below -5000ma. The robots could not survived for few seconds even with the low gain normal walk. After the improvements, the minimum current rUNSWift recorded was around -3500ma.

Unfortunately these improvements were not enough to prevent the battery overflow.

- Behavior modification

Lightning kick and forward kick are removed because they consume too much power when the robot gets up on the ground after finishing the move (section 4.3).

Before rUNSWift created the VRCOMM.CFG file, the robots crashed with any combination of the walk components. In general, movements with multiple walk components crashed more quickly than walks with singe walk component. For example, setting:

```
forward = 4
turn    = 4
left    = 0   (two walk components)
```

is more likely to crash the robots than setting:

```
forward = 8
turn    = 0
left    = 0   (one walk components)
```

After VRCOMM.CFG had been created, the robots were able to walk one dimensionally without crashing even they hitted

other objects. Unfortunately multiple dimensions walk still leaded to battery overcurrent. Since the older OPEN-R (OPEN-R SDK 1.1.5 r1) is no longer available, the two dimensional graphs can not be reproduced. These two dimensional graphs in figure 4.3 come from my memory, so they may not be accurate. Given a pair of values (eg: forward = 4 and turn = 4), the robot was instructed to walk with these values for two minutes. The pair of values were considered as "safe" if the robot did not crashed in this two minute interval. Shaded area marks the safe combinations. Notice that these values assume no robot charging or hitting with any object such as the field border.

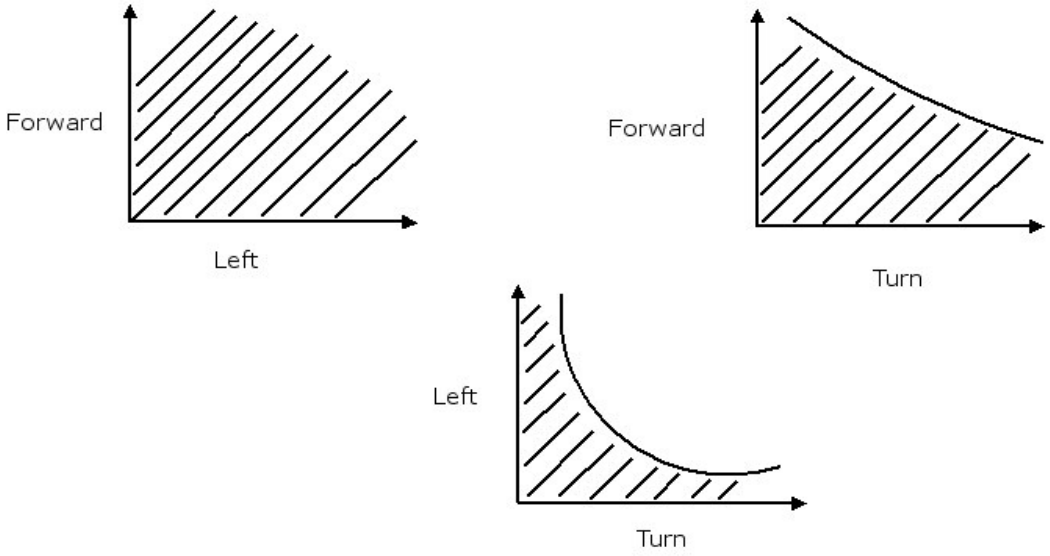


Figure 4.3: Two dimensionally graphs. Shaded areas indicate safe combinations - ie. the robot would not crashed from battery overcurrent. The values were recorded after VRCOMM.CFG was introducted. Notice that these graphs may not be accurate because they come my memory.

In figure 4.3, forward plus left component did not crashed the robot. Left plus turn combination crashed the robots more easily than other two dimensional combinations. Turning requires the motors to change the robot's heading and balance. Sideway walk requires less current than turning. Walking forward requires the least current, neither the heading nor the balance needed to be changed. In the Australian Open, forward, turn and left components were clipped to values within the shaded area of the graphs.

The way the robot chasing the ball (hover to ball) was modified to take advantage of the fact that one dimensional movement would never crashed and eliminated the left plus turn combination. Three dimensional movements were disabled, replaced by two dimensional walks. See section 5.8 for more details. The turning speed during getbehind ball was also reduced.

Modified the robot movement helped, the robots were able to survive much longer than originally. However it could not prevent the battery overflow specially when the robots were charging into each others.

4.2.5: Final solution

None of the method mentioned above could prevent the battery overcurrent. The final solution is the combination of the following methods:

- Low gain normal walk
- Locomotion improvements
- Behavior modifications.

Overall a single robot was able to play a whole ten minutes game without crashing. Unfortunately a full game (ie. four robots) still could not be played because the robot charging (leg lock) easily crashed the robots. The team could only lasted around four minutes. Although this year's competition has introduced the pushing rule, it is not possible to avoid physical contact with other robots.

Battery overflow was finally fixed by Sony's OPEN-R 1.1.5 r2 which was released few days before the Australian Open and dynamic gain.

4.2.6: Impacts on the behavior

Battery overflow had a significant impacts on the behavior.

- Very little behavior development could be done. The strategies and behaviors in the Australian Open were largely based on the previous year. Too much time was spent on investigating the

battery overflow.

- To prevent crashing, the robots were very slow. In the Australian Open, rUNSWift was the second slowest team. The slow walking speed was one of the major causes of failure in the Australian Open.

- rUNSWift was not able to develop new kicks for the ERS-7. Previous but now ineffective kicks must be used. For example, the lightning kick was replaced by chest push. Chest push is not an effective kick, its power is limited (page 146 of [13]).

4.3: Ball grabbing

4.3.1: ERS-7 ball grabbing

The ERS-210 robots are able to grab the ball with their front paws, this is not possible with the ERS-7. ERS-7's chest is larger than ERS-210's chest while ERS-7's paws are not longer than ERS-210's paws. As a consequence the offset between the ERS-7's chest and paws is too short for grabbing the ball with the paws. All of the previous years skills involve ball grabbing are not performing well for the ERS-7 robots. These skills include dribble, turn kick, visual avoidance opponent kick, lightning kick, front kick, these kicks have been rewritten or removed this year.

Figure 4.4: ERS-210 and ERS-7 grabbing with their front paws. Although not obvious here, the ERS-7 robots cannot grab and turn the ball with its paws.

Fortunately ERS-7 robots can still grab the ball with their front paws and head. ERS-7's head has an extra degree of freedom (crane), the extra degree allow the robot to move its head up and down without moving its neck. ERS-7's neck is longer than ERS-210's neck. According to the model information specifications [11], ERS-7's neck has a length of 80cm, while ERS-210's neck has a length of 40cm. ERS-7's head is also longer than ERS-210's head. ERS-7's head has a length of 76.9cm, while ERS-210's head has a length of 66.6cm [11]. As a consequence, ERS-7 robots are able to reach and grab the ball with their head moving forward.

Figure 4.5: ERS-7 grabs the ball with its head.

When the robot wants to do a ball grabbing, it moves toward the ball with hover to ball (section 5.8). When it is close to the ball, it stops, move its front paws and head forward. When the head is above the ball, the robot moves the head down. If the robot wants turn with the ball, it remain its ball grabbing stance and use the normal walk to turn on the spot.

In general ERS-7 cannot grab the ball reliability. Sometimes the robots hit the ball with their chest or head before the ball is grabbed, as a consequence the ball rolls out but the robots are not awared. The ERS-210 robots would immediately know that the ball has rolled out, because they can see the ball while holding it. Since ERS-7 needs to move its head down for grabbing, it cannot see anything other than the field.

Sometimes the mouth joint value and the body sensor can tell if the ball has been grabbed successfully. When the robot moves its head down, it opens its mouth. If the ball is below the head (ie. successful grabbing), the mouth may not be able to open completely, in this case checking the mouth's joint value can tell if the ball is grabbed. Also if the ball has been grabbed successfully, it would block the chest sensor, hence the value of the chest sensor should be much higher. Unfortunately this technique is not totally reliable because the readings are not consistent, but it is better than nothing. This technique was used in this year's competition. In this year's competition, no team could implement a perfect ball grabbing.

4.3.2: Future development

The future teams should replace the current ball grabbing with UTS's ball grabbing. In this year's world open, UTS robots grabbed the ball with their mouth. Although their robots still could not see the ball, they could see the front. This skill allowed the UTS robots to see the opponents and target goal while holding and moving with the ball. This skill would be very useful for visual opponent avoidance kick (section 5.9). Interested readers are referred to the UTS vs Germany 2004 Sony RoboCup Four-Legged League final video [23].

The current ball turning speed need to be improved in the future. In this year's competition, lots of teams could turn with the ball much faster than rUNSWift.

4.5: Turn kick

Turn kick was one of the major kick used in the last year strategies. It is a kick that can hit the ball 90 or 180 degree. Before the kick can be executed, the ball must be grabbed with the front paws. If the grab is successful, the robot would turn in the direction it wants to kick the ball. The kick is completed when the robot releases the ball and hit it with one of its front paws.

Turn kick is a fast kick. The robot can remain in walking stance throughout the kick, so less recovery time is required. Also the robot doesn't need to lie down, in contrast to the lightning and turning kick.

The time of ball release must be calibrated carefully. The calibration involves determining when to release the ball in a step cycle.

Figure 4.6: Turn kick. (Image courtesy of 2003 UNSW RoboCup report [13])

More details including when turn kick was used can be found in last year's thesis (page 209 of [13]).

Turn kick has been removed in this year's strategy. Reasons:

- ERS-7 robots cannot grab a ball with its front paws, hence they cannot turn and then hit the ball.
- The time to release the ball must be calibrated again for the new models since the locomotion module has changed. The calibration is very time-consuming — if the ball is released too early, the paw may miss it. If the ball is released too late, the ball may get trapped between the paws (page 212 of [13]).

The whole calibration process must be repeated when a significant change occurs to the locomotion module. rUNSWift couldn't afford to spend too much time on calibrating the turn kick specially a stable walk was not achieved until after the Australian Open.

Furthermore it was very difficult to calibrate when battery overcurrent occurred frequently.

Also rUNSWift ERS-7 robots cannot turn on the spot quickly. Turn kick would not be powerful with a slow turn.

Before the Australian Open, the strategies were based on the previous year. Turn kick was simply subtitled by sideway kick.

The left 90 degrees turn kick was replaced by right sideway kick and the right 90 degrees turn kick was replaced by left sideway kick. Sideway kick was selected because this was the only kick that could hit the ball sideway.

The 180 degrees turn kicks were replaced by sideway kick after ball turning. The calibration process for sideway kick with ball grabbing is easier than the calibration process for turn kick, since only the number of frames required for turning need to be calibrated. As long as the robot's turning speed remain unchanged, the frames required remain unchanged.

Subtitling turn kick with sideway kick did not work too well. Sideway kick cannot not hit the ball exactly 90 degrees to the side. A strategy would only be effective if all the skills are integrated carefully, simply replace a skill with another skill would not work. But replacing turn kick with sideway kick is still better than allowing the ERS-7 robots to perform turn kick.

After the Australian Open, a new attacker strategy was written. Turn kick was removed in the new strategy. If the robot wants to turn with a ball, it would perform ball grabbing with either the handkick or sideway kick. Refer to Chan K.C's thesis report [6] for more details.

4.4: Australian Open performance

In the Australian Open, rUNSWift was the second slowest team. The slow speed was one of the major failures in the Australian Open. The fast elliptical walk was not used. Fearing of battery overcurrent, the low gain normal walk was used in the first three rounds of the Australian Open. However none of the robot crashed, so the high gain normal walk was used against UTS. Surprisingly none of the robot crashed in the match versus UTS. No robots crashed because rUNSWift switched to the OPEN-R 1.1.5 r2 which was released few days ago before the Australian Open.

The previous year strategies in particular the team cooperation worked reasonably well. The ball grabbing and sidekick also worked reasonably well. rUNSWift robots were able to grab the ball most of the time, partly because the robots were slow. Ball grabbing plus sidekick was able to turn and hit the kick toward the target side reliably. However sometimes the red robots backoff from the ball because they recognized the orange ball as its red teammate. For more details, refer to Xu.J's thesis [8] and Lam.C.K's thesis [5].

The robots could play better if they had a better localisation. Often the robots mislocalised when they could only see one beacon. The goalie had a very poor localisation, it walked out of the goal box frequently. For more details, refer to Whaite.D's thesis [7].

Although some of the behavior skills were reworked, the overall strategies were not compatible for the new robots and new rule. For example the strategies being used did not consider the new pushing rule and ready state. The attacking speed was slow, chest push could not hit the ball very far away and sometimes the robot used the hover to ball when it was better to hit the ball with paw kick, maintaining speed. Also the kicking triggers were not tuned well, sometimes the robots did not kick the ball because of the relatively unreliable vision. Clearly the strategies and decision tree must be rewritten and this was done after the Australian Open.

Chapter 5
Behavior After Australian Open

5.1: Overview

This year the behavior module was rewritten after the Australian Open. Although the rewrite was time-consuming, it was necessary:

- The old strategies in particular the attacker strategies are designed for the ERS-210 robots. Simply modifying the old codes did not work, as seen in the Australian Open. For example integrating a new walk such as elliptical walk required the hover to ball to be rewritten, which had a significant impact on how the attacker approaches the ball.

- The 2004 rUNSWift team realized that maintaining the old code would be a nightmare. The 2003 rUNSWift codes are extremely unorganized and hard to understand. Most of the 2003 codes are uncommented. Even they are commented, sometimes the comments just do not make any sense. Since all the 2003 rUNSWift team members have left the team, so the new team must rewrite the codes.

- rUNSWift had only two months before the world open. To speed up the development, rewriting the behavior with Python was necessary.

- rUNSWift wanted to try some new strategies, such as no ball grabbing and directional paw kick. Refer to Chan.K.C's thesis report for more details [6].

- rUNSWift was not able to make significant behavior improvements before the Australian Open due to battery overcurrent. After it was fixed, the time for significant improvements came.

5.2: Porting process

This section describes the porting process in some details. The first step was to determine all the functions required by the Python behaviors. After they were determined, their accesses were given to the Python behaviors. Function requires are vision functions (eg: ball distance), localisation functions (eg: self position), wireless functions (eg: teammate position) and locomotion functions (eg: the current joint values).

The team then started to port the fundamental behavior skills, such as the ball tracking, hover to ball and active localise. Ball tracking allows the robot to track the ball whenever it sees the ball. The hover to ball skill is described in section 5.8.

After the fundamental skills were completed, the basic skills were then ported. These skills include get behind ball, paw kick, locate ball and dribble, they require the fundamental skills such as the ball tracking, hover to ball and active localise. Get behind ball allowing the robot to move behind the ball, facing toward the target side. Paw kick allows the robot to kick the ball with one of its front paw. Locate ball is a ball searching routine, the robot spins itself to look for the ball. Dribble allows the robot to push the ball toward the target goal. Get behind ball, paw kick and locate ball are same as the previous year, the only difference is that they are now written in Python, for more details refer to the previous year thesis report [13]. The dribble has modified slightly compared to the 2003 version, mainly the way how the robot grabs and turns with the ball. Refer to Chan.K.C's thesis report for more details [6].

Determining ball source and ball search were also ported into Python. Since they are same as the previous years, interested readers should refer to [13] for more details. Basically determining ball sources involve deciding which ball source should the robot choose, there are three choices: vision, wireless and gps. In general vision ball is the best, hence if the robot can sees the ball, it chooses the vision ball source. Otherwise the robot choose wireless ball source if it is reliable, information related to the wireless ball is reported from the teammates. However if vision and wireless ball sources are not available, the robot uses its world model to determine the ball position. Ball search involves looking for the ball. If the ball source is vision, then the robot simply tracks the ball with ball tracking. Otherwise

the robot quickly looks at the position where the ball is last seen, if the robot still cannot see the ball it tries to look at the wireless and gps ball position. If the robot still cannot find the see the ball, it goes into locateball routine — spin itself to look for the ball.

The next step was to port all the kicks into Python. This was done by allowing the Python modules to play back the predetermined joint angles.

After all the behavior skills and kicks were completed or at least almost completed, the team started to rewrite the higher level behavior skills and strategies. Decision trees (section 5.3), attack strategies, team cooperation (section 5.13), bird of prey (section 5.10), stealth dog (section 5.5) and goalie (section 5.11) were redesigned, modified and improved. A new player was also written — ready player (section 5.12).

The new attacking strategies use the new designed decision tree structure and new DKD (section 5.15). Several new strategies have tried such as directional paw kick and no ball grabbing. Directional paw kick allows the robot to hit the ball with its paw from/to any direction, hence maintaining fast attacking speed. Directional paw kick is designed to replace ball grabbing, since ball grabbing is in general not reliable and it is slow. Refer Chan.C.K's thesis for the new attack strategies.

Some new kicks were developed after the Australian Open, refer to section 3.5 for more details. Few days before the world open begun, visual opponent avoidance kick was invented, refer to section 5.9.

During the porting process, the codes were redesigned, modified and simplified such that the system is more manageable and understandable than the previous years.

The following sections describe the skills and strategies in more details.

5.3: Decision tree

5.3.1: Introduction

The rUNSWift behaviors are specified in a large hierarchy decision tree. A decision tree is a structure where nodes represent tests on one or more attributes. Each node considers a number of factors, such as robot's states, robot's world model, properties of different objects, wireless information and joint values. The output of a non-terminal node is an input of the next node. The output of a terminal node defines the next set of actions to be performed by the locomotion module. See figure 5.1.

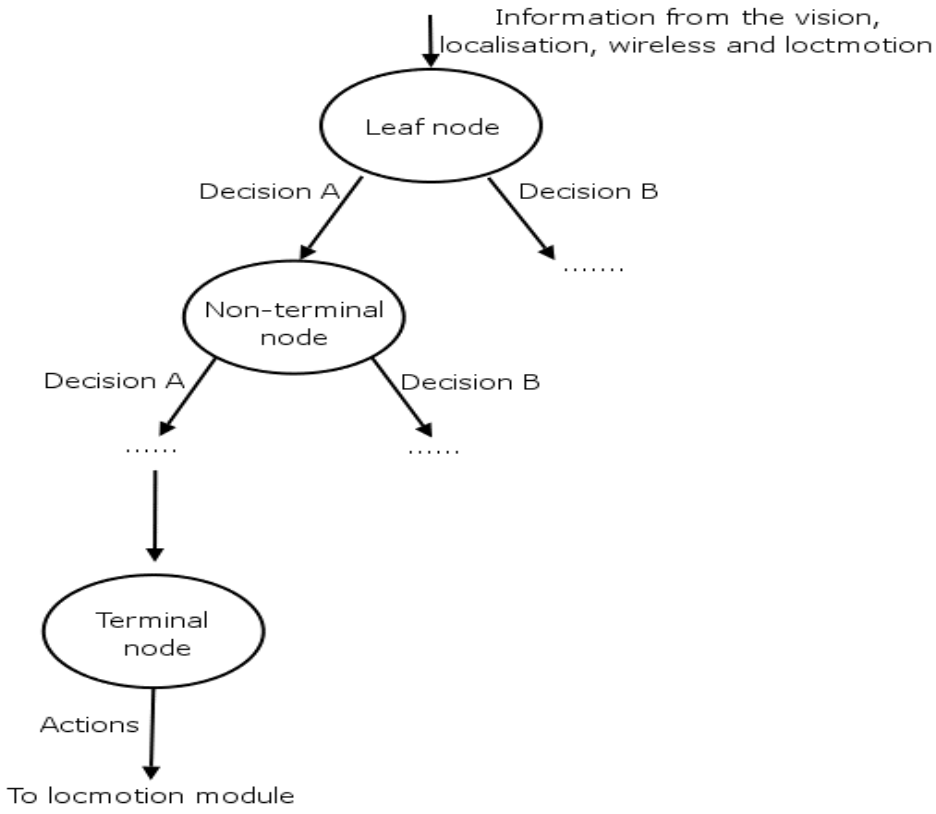


Figure 5.1: rUNSWift behaviour hierarchy decsion tree.

5.3.2: 2003 decision tree

The 2003 decision tree was splited into three levels: strategies, roles and skills (page 119 of [13]). See figure 5.2.

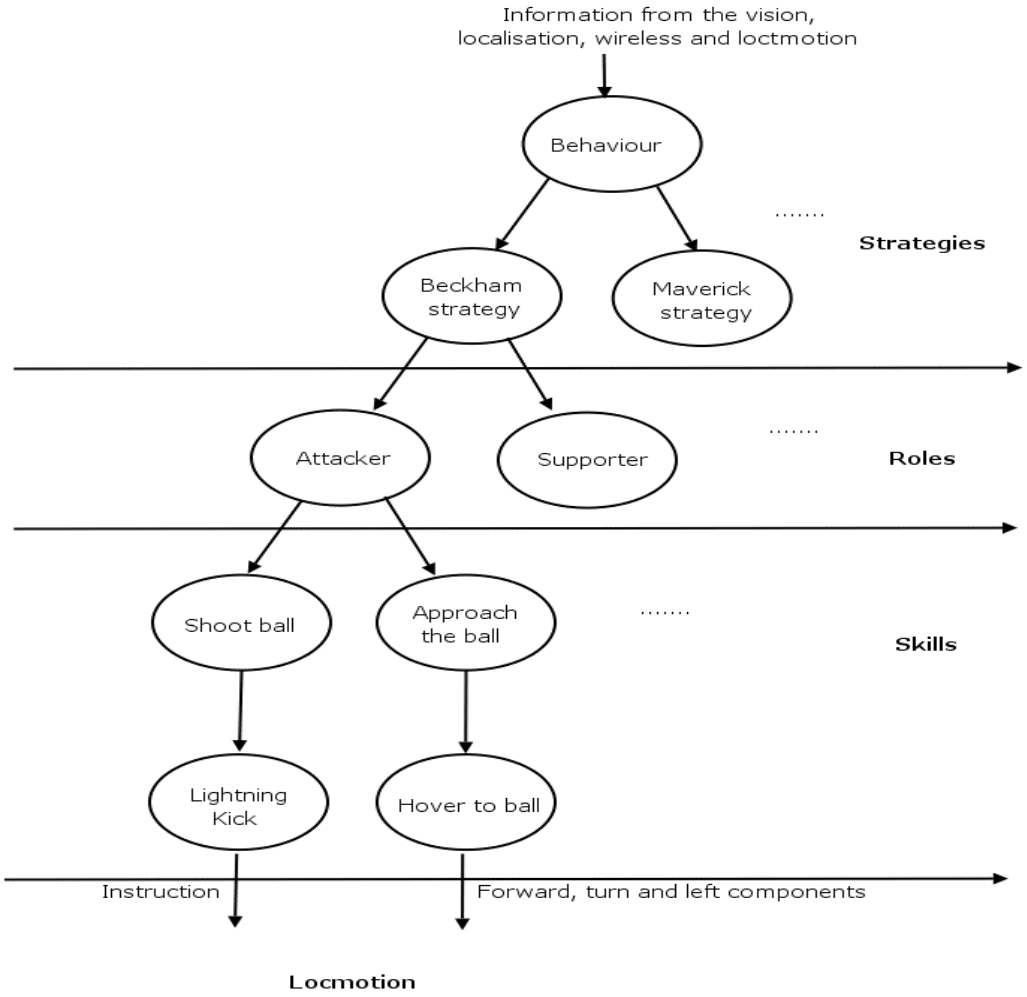


Figure 5.2: rUNSWift 2003 behaviour hierarchy decsion tree.

The strategies level defined a number of strategies that could be used. For example one strategy could instruct the robots to attack the ball simultaneously. Another could instruct the robots to defense. The strategies could change dynamically.

The roles level assigned roles to the robots. In the 2003 competition, most of the time rUNSWift assigned one forward as an attacker, one forward as a supporter and one forward as a striker. The roles could changed dynamically, known as dynamic role assignment. For more details, refer to the previous year rUNSWift thesis (page 121 of [13]).

The skills level defined a number of skills the robots could use. This level gave instructions to the locomotion module.

The top two levels in the hierarchy (strategies and roles) are flexible, dynamic and easy to maintain. However the skills level is complicated, and hard to maintain. Apparently the rUNSWift 2003 team copied and pasted the codes, a skill could appeared several times in the sources.

More importantly the decision trees defined in the skills level doesn't work very well for the ERS-7 robots, for example the shooting strategies. In 2003, the robot decided which kick it should be used only when the ball was close and under the robot's chin. When the ball is under the robot's chin, it decides its kick based on the world model. See figure 5.3.

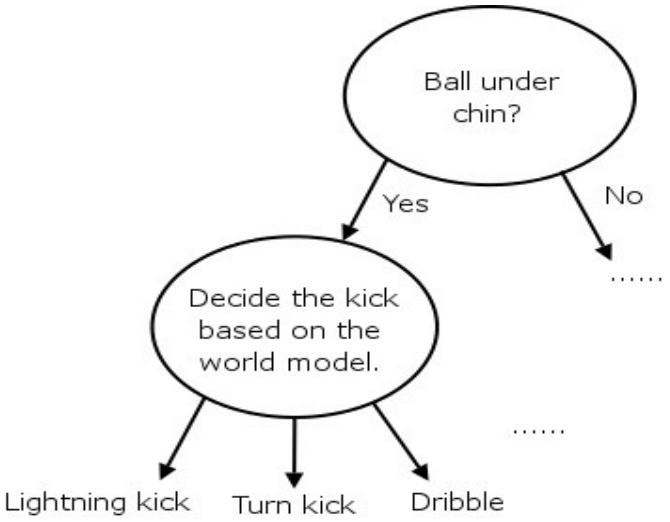


Figure 5.3: 2003 shooting decision tree.

Unfortunately not all the kicks share the same triggers. Some kicks are effective only when the robot is not too close to the ball, while other kicks may only be effective when the robot gains possession of the ball.

The 2003 decision tree had a number of lock modes. When a lock mode is activated, it forces the robot to do a particular action, skipping some of the decision paths. See figure 5.4.

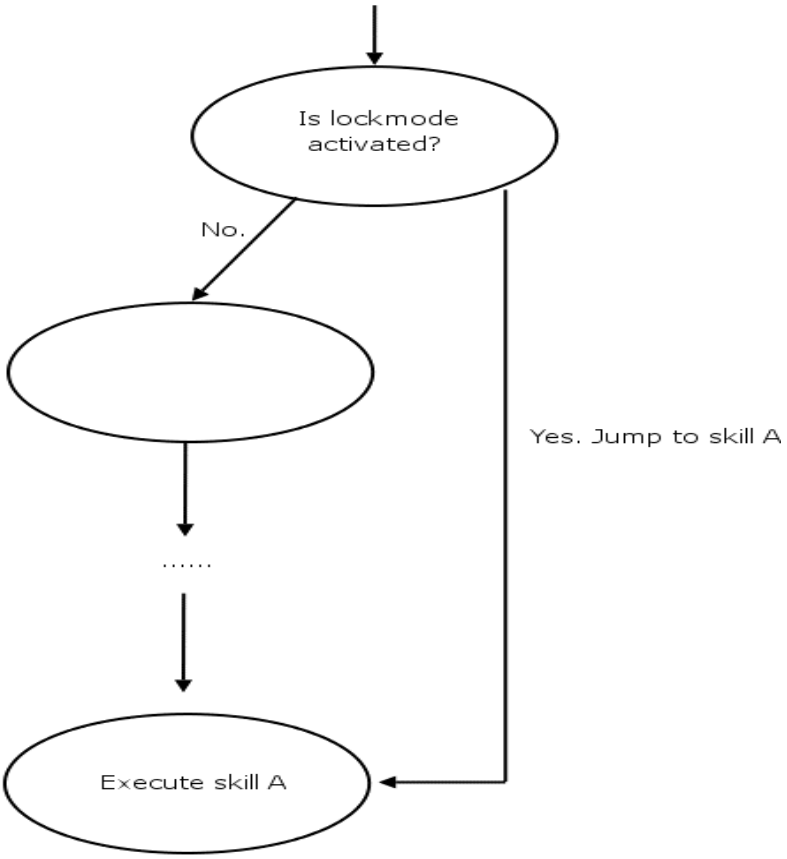


Figure 5.4: Lock mode.

Lockmodes could be useful for some high level skills such as dribble. The dribbling process requires more than one vision frame to execute, lockmode allows the program to "lock" the dribble until the robot completes it.

However lockmodes are evil, they make the code very hard to understand, debug and maintain.

5.3.3: 2004 decision tree

This year, the decision trees are partly rewritten in Python. The strategies and roles levels remain unchanged. The strategies level is also known as the players level. Python behavior allows rUNSWift to create different players, each with a different strategy.

The decision trees in the skills level were redesigned. Firstly skills which appeared several times in the sources were grouped together.

Currently each kick has a different set of triggers. The decision tree checks the triggers one after another and execute the kick if its triggers are satisfied. See figure 5.5.

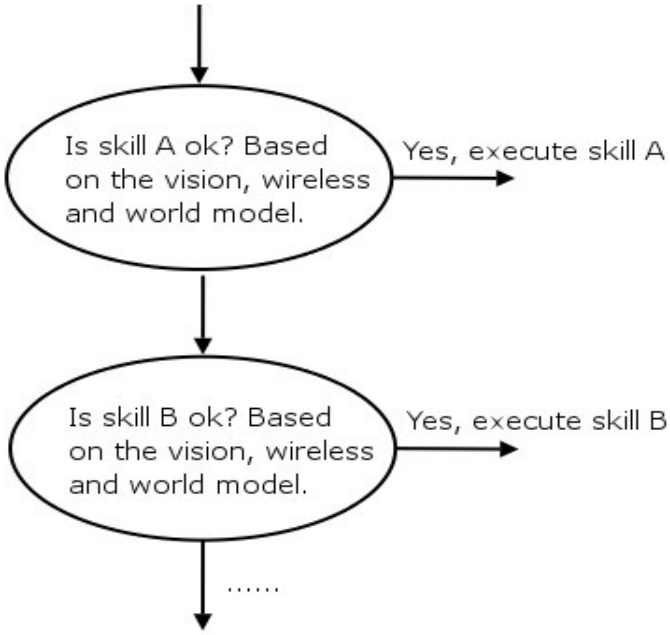


Figure 5.5: New decision tree.

The new decision tree provides flexibility. Changing the triggers for a particular skill is easy. Previously the changes might affect other kicks since all the kicks shared the same triggers.

An idea involved comparing the triggers simultaneously was also tried. Instead of comparing the triggers one after the another, they are all compared simultaneously and the final decision is based on how many triggers have been satisfied and the priorities of the triggers. For example if we have four skills: skill A, B, C and D. Skill A has the highest priority. Assume the triggers for skill A and D are satisfied, the robot knows it could perform skill A or skill D. Since skill A has a higher priority, the robot performs skill A. See figure 5.6. This comparison provides a great flexibility. This idea was implemented but not

used in the competition, since not enough time to tune it.

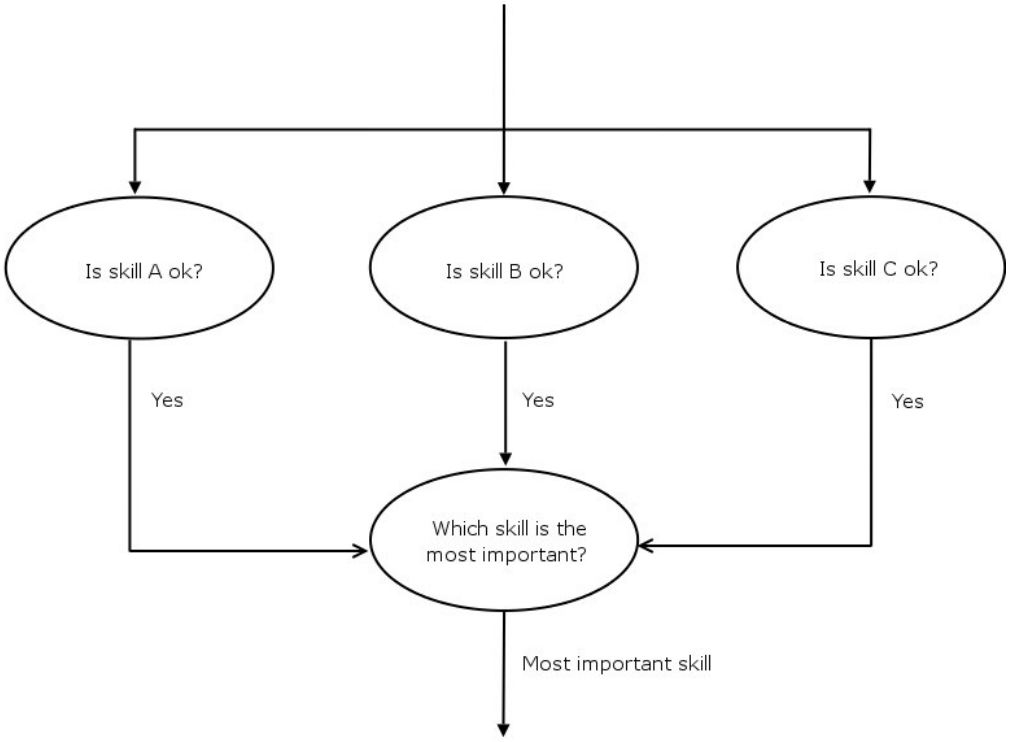


Figure 5.6: An experimented decision tree. Check the triggers simultaneously.

Lockmodes were introduced because the decision tree did not remember its previous decisions. Ideally the decision tree should makes a decision based on its previous decisions and current information. In theory, if the decision tree knows its previous decisions, it should be able to repeat its previous decisions without lockmodes. This idea was tried this year, but there was not enough time to complete it before the competition.

While the new decision tree is more flexible and maintainable, it is not tuned as well as the previous year due to limited development time. The future rUNSWift teams should refine the current decision tree.

5.4: Python behavior

5.4.1: Python advantages

This year the behavior module was completely rewritten after the Australian Open with Python. Previously it was written in C++.

C++ is a general purpose object-oriented programming language [30]. It compiles the C++ codes directly into a form that the machine can understand such as assembly language or binary code. Unfortunately all the C++ codes must be recompiled everytime a change occurs. This process is slow, the compile process often takes for few minutes. The programmer must patiently wait for the compile process to finish before the code can be transferred to a memory stick and a robot. If the programmer wantes to change something again, for example changing a constant, the robot must be shut down and all the C++ codes must be recompiled again.

Too much time is wasted for compiling C++ codes, it would be a better idea to use those time for actual code development. rUNSWift had only two months to prepare for the world open after the Australian Open, Python was introduced for speeding up the code development.

Python is an object-oriented scripting language. Because Python is interpreted, Python modules are loaded and reloaded dynamically, when imported by a running program [31]. Reloading allows a programmer to change parts of running programs without stopping.

This year rUNSWift decided to replace the high level (behavior) C++ code with Python. Python is selected because:

- **Dynamic Reload**

  Python allows a programmer to modify the behavior dynamically. When the behavior need to be modified the programmer sends the modified Python code to the robot by ftp. The Python interpreter dynamically reloads the new code. It is not necessary to recompile the Python code, also the robot doesn't need to be shut down while reloading. Development speed significantly improve with dynamic reload. Python allows

rUNSWift to modify the strategies dynamically.

rUNSWift has developed a tool known as SimpleRobotCommander to transfer the Python codes to the robot dynamically, see figure 5.7.

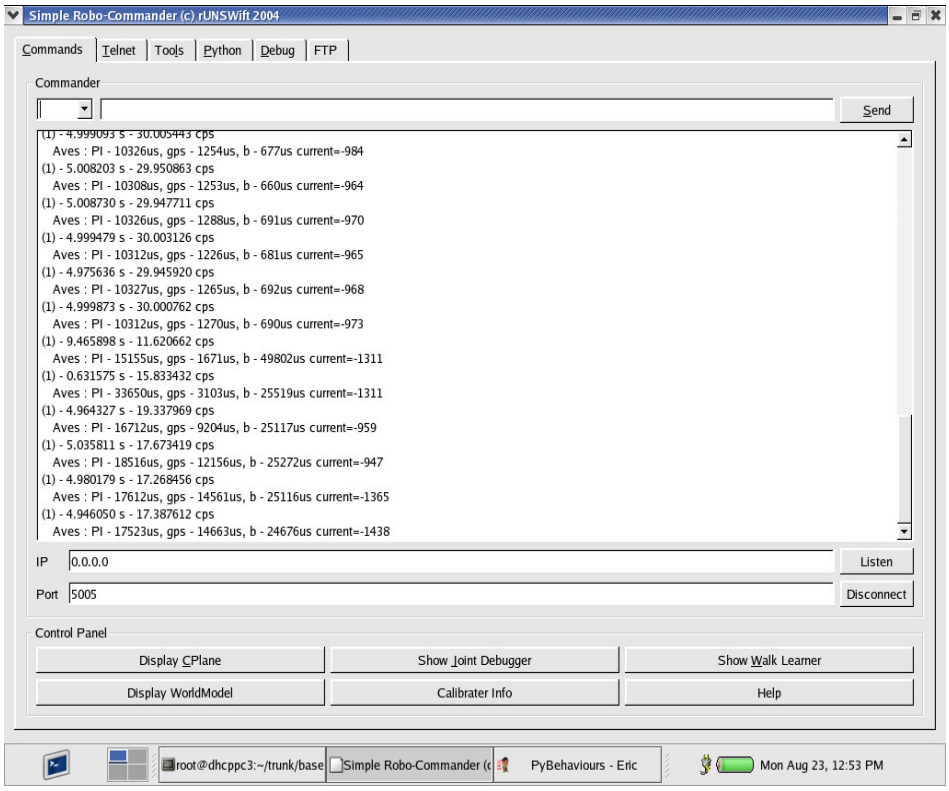


Figure 5.7: SimpleRobotCommander.

- **Robot crash**

Before Python was introduced debugging software crashe was not easy. If a C++ program crashes, the robot also crashes, hence the programmer may not able to spot the bugs quickly. *Crashing Tool* developed by Griff University is able to report a stack of functions and assembly codes [34]. However it is not able to report the line the robot has crashed. Also this tool usually generates more than one stacks, they must be all examined.

Since Python programs are controlled by an interpreter, the robot would not crash when the software crashes. Instead the interpreter reports useful information about the crash including

a stack of functions and lines including the line the robot has crashed from telnet. The programmer can simply examine the stack, modify the code, transfer the code to the robot with ftp and dynamically reload it. No need for recompiling or shut down the robot. Debugging Python programs is much easier than debugging C++ programs.

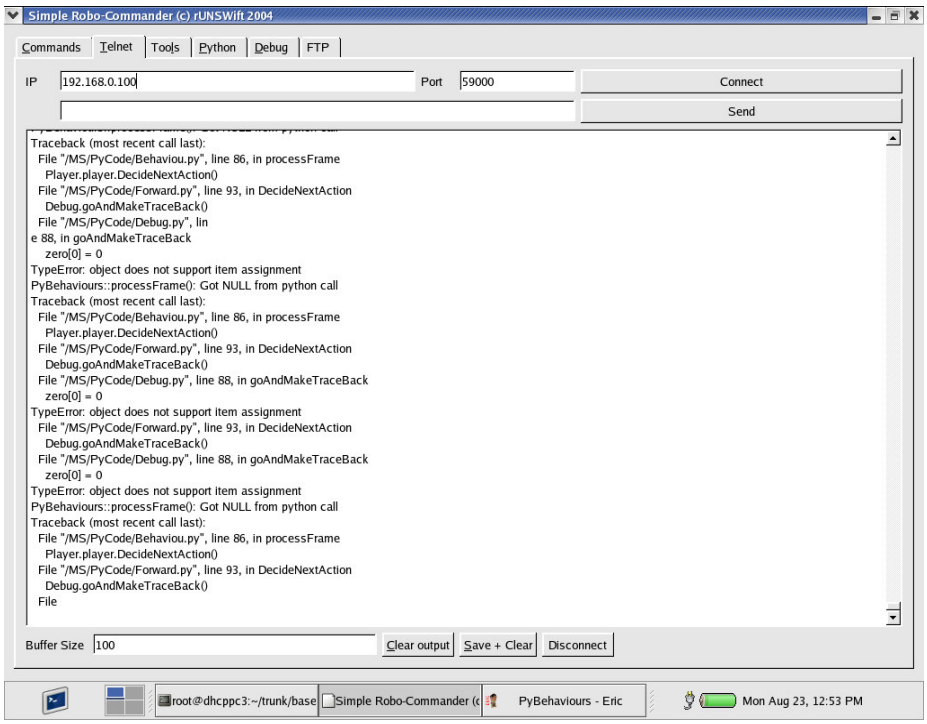


Figure 5.8: Python stack.

- **Dynamically evaluating variables and expression**

Variables and expressions can be dynamically evaluated with Python. No need to modify or recompile the codes, simply enter the variables or expressions in SimpleRobotCommander, dynamically reload is not necessary.

For example given a very simple Python function below:

```
def function:
        variableA = 1
        variableB = 2
        variableA = variableA + variable B
        # What is the value of variableA now?
```

If the programmer wants to know the value of variableA after the calculations, the programmer can modify the code and dynamically reload it.

```
def function:
    variableA = 1
    variableB = 2
    variableA = variableA + variable B
    # Output the value of variableA
    Print variableA
```

However the reloading process takes time — time for ftp transfer plus time for Python to reload. It would be much better to evaluate the variableA with SimpleRobotCommander like in figure 5.9, no code change involve.

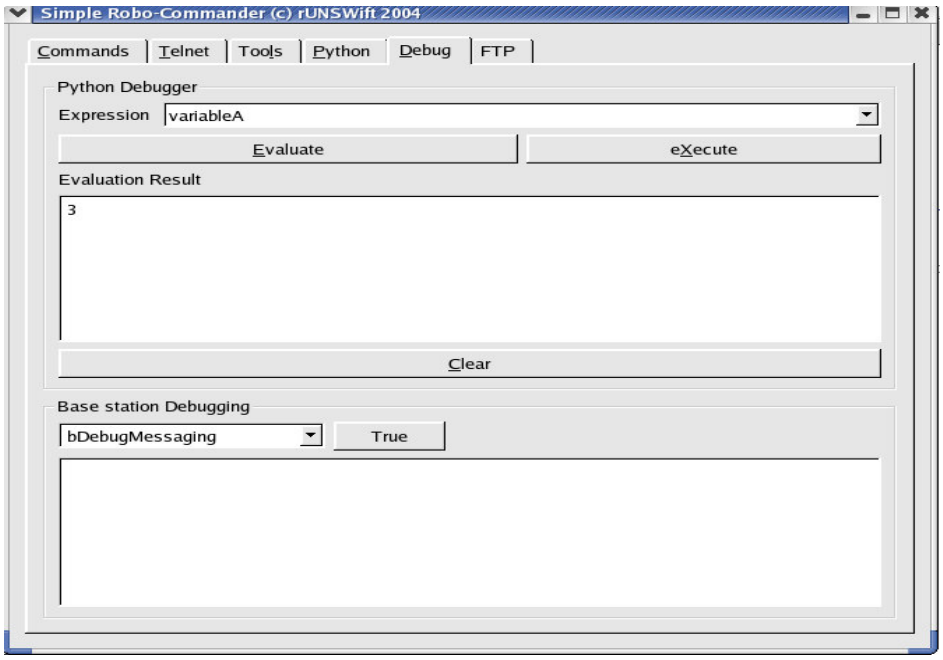


Figure 5.9: SimpleRobotCommander dynamically evaluating variable.

- **Dynamically executing functions**

Python functions can be dynamically executed. If a programmer wants to execute a function, instead of modifying the code, the function can be executed directly with SimpleRobotCommander.

- **Maintainable**

  Python code is typically 1/3 to 1/5 the size of equivalent C++ code. Python code is pseudo-code, it is easy to lean and maintainable than C++ codes [27], very important for rUNSWift since the team members change every year.

Currently the Python behavior receives information from the vision, localisation, locomotion and wireless modules, all written in C++. Python behaviors process the information in a decision tree and decide what would be the next action. The action is sent to the hardware through the locomotion module. Python behavior also sends wireless commands through the wireless module.

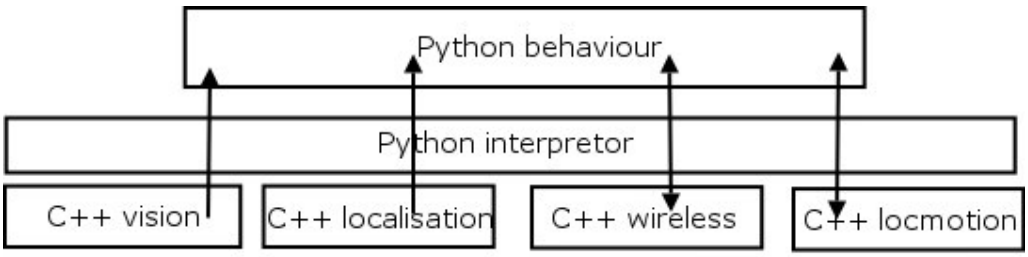


Figure 5.10: Relationship between python behaviour and low level modules written in C++.

5.4.2: Python disadvantages

Because Python codes are not compiled into assembly or machine code, they are processed by a Python interpreter, the execution speed is slower than C++ codes. Refer to [27] for more details. Also information from low level modules need to pass the Python interpreter, this information passing is slow.

Currently rUNSWift encounters occasional frame dropping. Even the improvements give by the disjoint sets blob algorithm (chapter 2) is not sufficient to allow the robot to execute Python behavior at full frame rate. C++ behaviors can be executed at full frame rate.

An experiment involved comparing the running time of vision, localisation, C++ behavior and Python behavior was tried. See figure 3.10 for the result. During the experiment, a robot was put onto the field for playing. It executed the C++ behavior which was used in the 2004 Australian Open. After the C++ behavior was finished, it executed the Python behavior which was used in the 2004 RoboCup world open. The execution times were recorded.

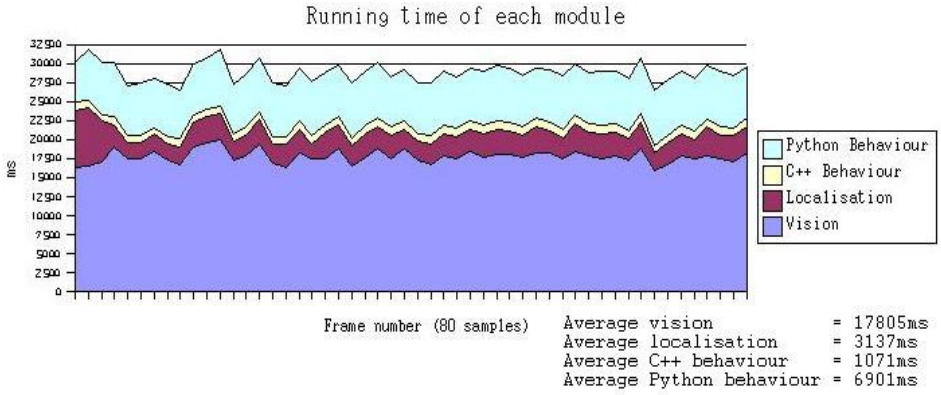


Figure 5.11: Running time of the modules.

Notice that behavior modules were executing almost constantly. This is because the decision tree has a constant running time. Behavior's running time is relatively independent of other modules.

The running time of vision module is depend on the camera image — as the image becomes more complex and noisy, the running time increases. The camera images in the experiment were medium quality. As seen in the figure, vision is the slowest module. Python behavior is the second slowest module, while the C++ behavior is the fastest module. As seen in the figure, Python behavior executes much slower than C++ behaviors (6901/1071 = 551%).

5.4.3: Conclusion and discussion

Python behaviors worked very well this year. rUNSWift was able to improve the system significantly in only two months after the Australian Open and before the world open. Lots of skills were written and invented in Python. The development speed with Python was much faster than with C++.

Currently a small parts of the behavior is still in C++ because the team didn't have time to port it into Python. The future teams will probably port the whole behavior module into Python.

However Python is slow — approx 5 times slower than C++. While behavior will probably stay in Python in the future, Python is not recommended for other modules in particular the vision.

The future teams definitely need to do something to cut down the current execution time. There are two possible solutions:

- Reduce vision's execution time. The vision will probably be rewritten in 2005. The next year team may need to reimplement and improve the color segmentation and blob algorithm, which contribute most of the vision execution time.

- Currently the low level information is passed to Python individually. Information passing is slow, it would be a better idea to pass all the required information simultaneously. See figure 5.12.

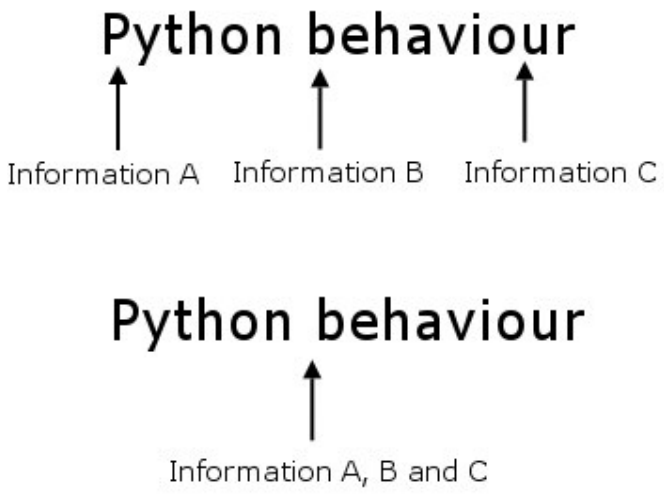


Figure 5.12: Different ways to pass information from C++ to Python.

5.5: Stealth dog

5.5.1: Overview

Stealth dog is an important skill. It allows the robots to take a curve path avoiding opponent robots. This skill is very useful for ball chasing when the opponents are in the way. Our robot doesn't want to charge the opponent and taken off the field for thirty seconds (pushing rule). A better idea is to walk past the opponents and attempt to reach the ball before the opponents.

5.5.2: 2003 stealth dog

Obviously taking a curve path toward the ball is slower than walking straight to the ball. Stealth dog should only be used when it is not possible or too difficult to reach the ball by walking straight to the ball.

In the previous strategy stealth dog was activated when:

- The robot could see the ball and it was far away. If the robot could see the ball, it would performed ball searching. If the ball distance was close, then there might not be enough time for the robot to follow a curve path and still be able to reach the ball before the opponent. In this case the robot should walked straight toward the ball and fought with the opponents.

- At least one opponent must be visible. The idea of stealth dog is to walk past the opponents. If only the teammates were seen, then the robot should backoff and allowed the teammates to gain possession of the ball.

- The robot drew a curved locus before stealth dog was activated. Stealth dog was activated only when the robot believed the locus would not hit an edge and hence slowing down the robot's movement.

- The opponent was closer to us than the ball is. If our robot was closer, then there was no point to follow a longer path. Simply walking straight toward the ball and gaining the control of the ball before the opponent.

- The opponent had not controlled the ball. If the opponent had controlled the ball, instead of following a long curve path, a better idea would be to walk straight to the ball and fought for the ball control. The robot believed the opponents had controlled the ball when the distance between the opponent and the ball was small.

- The opponent was obstructing our robot. The opponent was considered as obstructed when it was close and in front of our robot. This condition prevented the robot to stealth when it could walk straight toward the ball.

Since localisation and wireless may not be reliable, only the vision information is used in the conditions.

If the stealth dog is activated, the two closest opponent headings are calculated. In the event of only one opponent is seen, these two headings are equal. The average of these two headings are calculated. Stealth dog is a very simple skill, all it does is maintain a fix angle away from the average of the opponent headings. In 2003, this fix angle was 45 degrees (page 180 of [13]).

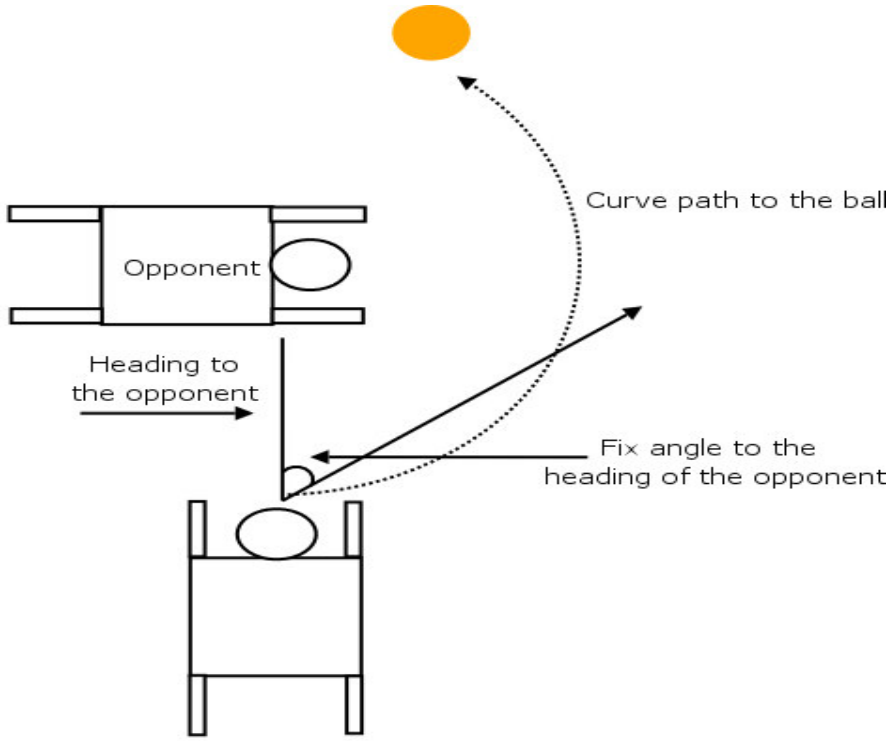


Figure 5.13: Stealth dog. It takes a curve path around an opponent toward the ball.

5.5.3: 2004 stealth dog

Stealth dog is used in this year's attacker and ready player strategies. A new stealth dog has been added — sideway stealth dog. The old stealth dog is known as curve stealth dog.

Curve stealth dog's implementation is identical as the previous year, it is used in this year's attacker and ready player. Only the trigger conditions for the ready player are changed slightly. Previously stealth dog only activated when a robot sees the ball, its turning direction is the direction to the ball. Ready players do not need to track the ball, so all the trigger conditions related to the ball are removed for the ready player. Also the direction of the curve is the direction to the ready player's kickoff position.

Sideway stealth dog has been introduced for the ready player. There old stealth dog has a drawback — it cannot be used when an opponent is too close, since there would not be enough room for the robot to take a curve path. See figure 5.14.

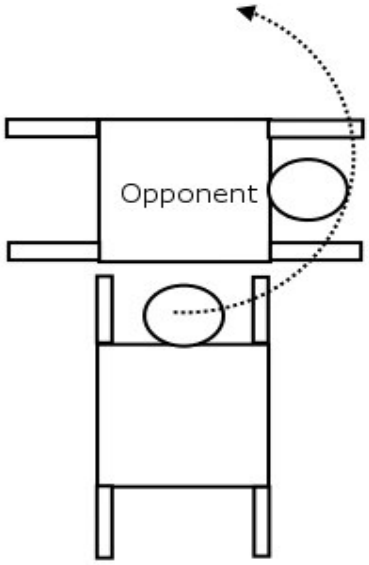


Figure 5.14: A problem from the previous year's stealh dog. The robot cannot walk past an opponent when it is too close. The robot doesn't have room to take a curve path.

Sideway stealth dog allows the robot to walk past the opponents even if they are extremely close to the robot. Instead of

taking a curve path, sideway stealth dog walks sideway until the robot cannot see any opponent or the opponents are no longer obstructing. See figure 5.15.

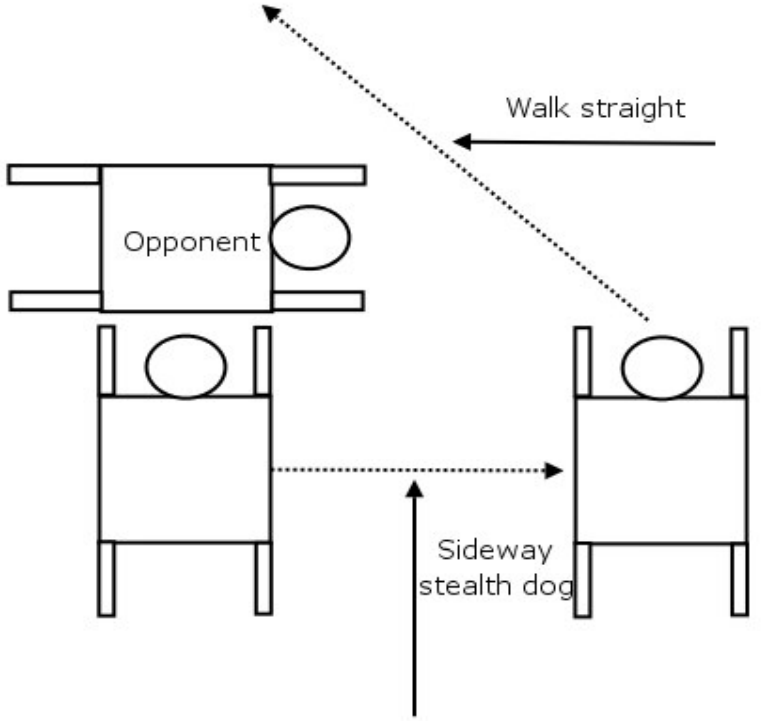


Figure 5.15: Sideway stealth dog. The robot moves sideway until it cannot see the opponent. After sideway stealth dog is deactivated, the robot walks straight toward its destination.

The trigger conditions for the sideway stealth dog are identical to the conditions for the curve stealth dog, except for the fact that the opponents must be very close to the robot. For the competition, this distance was set to 30cm. The direction the robot moves sideway is the direction to the kickoff position.

5.5.4: Conclusion and discussion

Curve and sideway stealth dog has performed well in the competition. Ready player was able to evade opponents most of the time.

Unfortunately stealth dog's performance is limited due to the

poor robot recognition. Sometimes the opponents cannot be recognized or only able to be recognized for few frames in particular the blue opponents. The robot distances reported by the robot recognition is also a problem, the distances are not accurate in particular when the opponents are close. Since sideway stealth dog is activated only when the opponents are close, the distance inaccuracy sometimes activate the sideway stealth dog when it should not be activated and sometimes deactivate when it should be activated.

Sideway stealth dog is slow because only the normal walk allows the robots to move sideway. In the future, a faster sideway walk type should be invented.

In this year's competition, sideway stealth dog was used only for the ready player. But it can be used for the attacker too. UTS did exploited the potential of sideway stealth dog in the competition. In the competition, when UTS robots saw opponents in front of them, they grabbed the ball, moved sideway until they couldn't see the opponents and then shot.

5.6: Stuck detection

5.6.1: Overview

Stuck detection is a technique allowing the robots to detect obstacles without actually looking at it. It is a new technique. Detecting obstacles is important because:

- The new rule specifies that the pushing robot must be taken off the field for 30 seconds. Stuck detection allows the robot to detect if it is pushing against other robots. Vision alone may not be adequate because the camera image is usually noisy when the robot is close to another robot. Also the robot may not see the robots it is pushing.

- The ready player needs the stuck detection to field borders.

Two stuck detection methods are tried — PWM duty cycle and obstacle scanning.

5.6.2: PWM duty stuck detection

PWM stands for Pulse Width Modulation, it is a way of digitally encoding analog signal levels. A sequence of duty cycle of digital square waves is used to encode an analog signal. Interested readers are referred to [32] for more details.

Each joint has a PWM duty value. When the joint is jammed (eg: something block the robot's way), PWM value should be much higher.

The PWM duty stuck detection algorithm is very simple. The algorithm takes the PWM value as an input and compares it with a calibrated PWM threshold. An obstacle is detected if the given PWM value is higher than the threshold. For the competition, the threshold is set to 1300.

Since most of the time the robot walks forward, the PWM of the front paws are used in the algorithm. However the PWM values report by the hardware are not consistent. Sometimes the PWM values rise above the threshold even the joint are not jammed. To minimise the error, the PWM value passes to the stuck detection algorithm is the maximum PWM value of the front paws over the last two seconds. The two second interval is a calibrated interval. From experiments, if the interval is lesser than two seconds, then there are not enough PWM samples to minimise the error. If the interval is greater than two seconds, then the robot reacts too slowly. For example if the interval is four seconds, when the robot hits an object, the instantaneous PWM rises above the threshold, but the maximum PWM value would only rise above the threshold four seconds later.

All the kicks require the leg joints to move quickly. Hence everytime a robot performs a kick, its leg PWM values always rise above the threshold. The algorithm takes care of this by counting the number of frames since the last kick has been performed. If the PWM value is higher than the threshold but the number of frames since the last kick has been performed is less than 150 (approx 5 seconds), then the algorithm assumes the high PWM values come from the kicking. See figure 5.16 for the code fragment.

```
/*
   Copyright 2004 The University of New South Wales (UNSW) and
   National ICT Australia (NICTA). This file is part of the 2004
   team rUNSWift RoboCup entry. You may redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of
   the License, or (at your option) any later version as modified
   below.  As the original licensors, we add the followin
   conditions to that license: In paragraph 2.b), the phrase
   "distribute or publish" should be interpreted to include entry
   into a competition, and hence the source of any derived work
   entered into a competition must be made available to all parties
   involved in that competition under the terms of this license.
   In addition, if the authors of a derived work publish any
   conference proceedings, journal articles or other academic
   papers describing that derived work, then appropriate academic
   citations to the original work must be included in that
   publication. This rUNSWift source is distributed in the hope
   that it will be useful, but WITHOUT ANY WARRANTY; without even
   the implied warranty of MERCHANTABILITY or FITNESS FOR A
   PARTICULAR PURPOSE.  See the GNU General Public License for more
   details. You should have received a copy of the GNU General
   Public License along with this source code; if not, write to the
   Free Software Foundation, Inc., 59 Temple Place, Suite 330,
   Boston, MA  02111-1307  USA
*/
```

```
Def amIStuckBasedOnPWMDuty():

        # Get the maximum PWM value of the front
        # paws over the last two seconds.
       (maxPWM, _), = HelpLong.getStuckInfo()

       stuck = (maxPWM >= 1300 and
                Kick.framesSinceLastKicking >= 150)

       return stuck
```

Figure 5.16: PWM duty stuck detection code fragment in Python.

PWM duty is currently used in the ready player for detecting field borders.

PWM duty stuck detection is simple and effective. However it can only detect the stuck not evading it since the PWM value only rise above the threshold after the robot hits an obstacle.

5.6.3: Obstacle scanning stuck detection

Obstacle scanning stuck detection allows the robot to detect the obstacles before the robot hits them. This method involves the IR head sensor. The IR head sensor has two channels — near and far. The near sensor detects obstacles with distance between 5cm to 50cm. The far sensor detects obstacles with distance between 20cm to 150cm [11]. Only the near sensor is used since the minimum detectable distance for the far sensor is too far to be useful.

When the robot performs a scan, it would slow down or pause otherwise it may hits the obstacle. During the scan, the robot pan its head from left to right or right to left. The scan should be quick, not wasting too much time on it, the whole process should take less than few seconds to complete. The obstacle is detected when the distance reports by the IR sensor is less than a calibrated threshold. Unlike the PWM stuck detection, the direction to the obstacle can be determined, it is equal to the heading of the head. For the competition, the threshold is 20cm.

Obstacle scanning stuck detection works reasonably well, but it is not totally reliable. From the experiments, the distances give by the sensor are not consistent. Information from other sources is required to improve the reliably. For example if the robot wants to scan for opponent robots, the fact that the robot can see an opponent and the distance gives by the IR sensor is small suggest an opponent is blocking the robot's way.

Obstacle scanning stuck detection is used by the attacker. It is not used by the ready player since the ready player already has three types of obstacle avoidance (section 5.12.10). More details on how this algorithm is used in the attacker including what would the robot do if it detects a stuck can be found in Chan.K.C's thesis [6].

```
/*
   Copyright 2004 The University of New South Wales (UNSW) and
   National ICT Australia (NICTA). This file is part of the 2004
   team rUNSWift RoboCup entry. You may redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of
   the License, or (at your option) any later version as modified
   below.  As the original licensors, we add the followin
   conditions to that license: In paragraph 2.b), the phrase
   "distribute or publish" should be interpreted to include entry
   into a competition, and hence the source of any derived work
   entered into a competition must be made available to all parties
   involved in that competition under the terms of this license.
   In addition, if the authors of a derived work publish any
   conference proceedings, journal articles or other academic
   papers describing that derived work, then appropriate academic
   citations to the original work must be included in that
   publication. This rUNSWift source is distributed in the hope
   that it will be useful, but WITHOUT ANY WARRANTY; without even
   the implied warranty of MERCHANTABILITY or FITNESS FOR A
   PARTICULAR PURPOSE.  See the GNU General Public License for more
   details. You should have received a copy of the GNU General
   Public License along with this source code; if not, write to the
   Free Software Foundation, Inc., 59 Temple Place, Suite 330,
   Boston, MA  02111-1307  USA
*/
```

```
def obstacleScanning():

    global stuckDetectTimer

    irdist = VisionLink.getAnySensor(Constant.
             ssINFRARED_NEAR)

    # Pan the robot's head.
    scanForObstacle()

    stuckDetectTimer += 1

    if stuckDetectTimer == 30:
            stuckDetectTimer = 0

   # If the distance is small and an opponent is seen, then
    # there must be an opponent in front.
    if irdist < 200000 and HelpLong.canSeeOpponentWithinDist
       (50):

            ......


# Pan the robot's head from left to right or right to left.
def scanForObstacle(speed = 10, tilt = -10, crane
                    = -10):

      global panDirection, panx

      if panx >= 80 and panDirection == Constant.
         dANTICLOCKWISE:
            panDirection = Constant.dCLOCKWISE
```

```
elif panx <= -80 and panDirection == Constant.
     dCLOCKWISE:
      panDirection = Constant.dANTICLOCKWISE

if panDirection == Constant.dCLOCKWISE:
      panx -= speed
else:
      panx += speed

FWHead.compulsoryAction = FWHead.fixHead
FWHead.fixHeadAt = (panx, tilt, crane , Constant.
                    HTAbs_h)
```

Figure 5.17: Obstacle scanning code fragment.

The current robot recognition is not very good. Obstacle scanning offers an alternative to detect opponents.

5.6.4: Conclusion

Neither the PWM duty cycle nor obstacle scanning algorithm is totally reliable. They must be used with information from other sources, such as localisation and vision. For example, the ready player uses stuck detection, stealth dog (vision) and vector avoidance (localisation) to detect and evade obstacles.

5.7: Dynamic gain

5.7.1: Description

The new ERS7 models have two servo gains: standard (high) and weak (low) gain. A servo gain defines the PGAIN, IGAIN, DGAIN, PSHIFT, ISHIFT, SHIFT for each CPC primitive locator (Sony OPEN-R SDK ERS7 model information page 10). Since PSHIFT, ISHIFT and SHIFT are fixed values (page 10 of [11]), the differences between the low and high gain are PGAIN, IGAIN and DGAIN (PID). The PID defines how strong the motors should move, it has a significant impact on the robot performance.

Multiple servo gain is only introduced this year, the previous ERS-210 models have only one standard servo gain [33]. The robots can perform very powerful actions with high gain but these actions consume lots of energy and lead to battery current overflow (see section 4.2). Before the Australian Open, rUNSWift robots could not survive for over one minute with high gain. Hence high gain was not used before and in the Australian Open except the match against UTS (see section 4.5). To prevent the current overflow, slow gain must be used but the motion generates is too slow for being competitive. The slow walking speed generated by low gain was one of the major causes of failure in the Australian Open.

Fortunately servo gain can be changed dynamically, as Sony suggests, the robot should *use standard values normally, and use weak gain values when large vibrations occur* (page 10 of [11]). This technique is known as dynamic gain, it allows the robots to maintain their fast high gain speed while preventing the current overflow with low gain.

The latest OPEN-R has a high overcurrent threshold. However even the latest OPEN-R does not allow the rUNSWift robots to use the high gain at all times. Occasionally the robots encounter high battery current with high gain, if the robots still remain in high gain state, the hardware would automatically shut down the robots, preventing damages to the robots. Dynamic gain allows the robots to switch into low gain and avoid automatically shut down. Although low gain gives slower motion, but it is still better than pausing the robot or allowing the robots to be shut down.

Dynamic gain is a very simple technique. The software monitors

the battery current value every frame. The robot would always stay in high gain unless the battery current rises above a threshold, when this happen the software immediately switch to the low gain until the battery current drops below another threshold. The first threshold triggers the switching from high to low gain, the second threshold triggers the switching from low to high gain. The reason why two thresholds are introduced is that the robot has more time to cool down, since robot's current is accumulated (see section 4.2). Also current reading may not be accurate, the software should switch the gain back into high only when it is sure it is safe to do so. It is important to note all kicks are currently performing with high gain. Since the robots would not continually perform kicking, high gain kicking should be fine.

The current overflow threshold for ERS-7 is 3900ma [21]. However from practice games, the robots crashed frequently if the robots switched into low gain when their current is higher than 3900ma. Apparently the robots did not have time to switch into low gain and cool down. 3900ma is a hardware threshold, the software should not allow the current rises near it. For the competition, the software current overflow thresholds are set to 3000ma and 2300ma. See figure 5.18.


```
      # Set the motor join gain by battery current.
      def dynamic_gain ():
```

```
current = VisionLink.getBatteryCurrent()

# Force to high gain if the robot is performing a kick.
wt = Global.finalAction[Constant.AAWalktype]
      if wt in [Constant.ChestPushWT, Constant.
      FwdKickWT, Constant.FastKickWT, Constant.
UpennRightWT, Constant.UpennLeftWT,
Constant.DiveKickWT, Constant.
HandKickRightWT, Constant.HandKickLeftWT]:

            Global.forceHighGain = True

# If not forced to switch into high gain, execute the
# dynamic gain algorithm.
if Global.forceHighGain is None:
      if current < 2300 and not Global.isHighGain:
            setHighGain()
      elif current > 3000 and Global.isHighGain:
            setLowGain()
else:

      if Global.forceHighGain :
            setHighGain()
      else:
            setLowGain()
```

Figure 5.18: Dynamic gain code fragment. Written in python.

Dynamic gain was useless before the release of OPEN-R 1.1.5 r2, since at that time even the most stable low gain normal walk would crashed. Clearly dynamic gain is useless if the robot stays in low gain most of the time. Fortunately this is not the case, the robots remain in high gain state most of the time during the game.

5.7.2: Conclusion

Dynamic gain worked very well in this year's world open - rUNSWift didn't experience any hardware crash. Undoubtedly dynamic gain will stay in the future.

5.8: Hover to ball

5.8.1: 2003 hover to ball

Hover to ball is one of the most fundamental behavior skill. It directs a robot to move toward the ball as quickly as possible. Usually this skill directs the robot to walk straight toward the ball.

In 2003, this skill was based on the turn factor and the ball distance. The turn factor was directly related to the ball heading, the 2003 formula:

$$turn = ball\text{-}heading \ / \ 2$$

Two walks were used - canter and offset walk. Canter walk allows reliable turning but it is slow. Offset walk is fast but it cannot turn quickly. The hover to ball decisions involve deciding which walk would be better to use and what would be the maximum forward and left component. The decision:

- If the turn factor is small compared to the distance to the ball then the faster offset walk would be used. Offset walk is ideal because the far distance to the ball should provide enough time for the robot to turn even with the offset walk. In this case the left component is set to zero and the forward component is set to the maximum forward speed.

- Otherwise, the turn factor is large compared to the distance to the ball, then the slower canter walk would be used. Canter walk allows the robot to turn smoothly which is not possible with the offset walk. The forward and left component are depend on the turn component. See figure 5.19 for values. These values were calibrated by the 2003 rUNSWift team. Notice that the values are also depend on the turning direction since the robot's body is not symmetry (page 222 of [13]).

TurnCCW t	Max Forward	Max Left
$0 \leq t < 10$	5.5	5.5
$10 \leq t < 20$	4.5	4.5
$20 \leq t$	3	2
$0 > t > -10$	5	5
$-10 > t$	1	3

Figure 5.19: 2003 hover to ball parameters (Image courtesy (Image courtesy of 2003 UNSW RoboCup report, page 223 of [13])

Hover to ball slows the robot when it is close to the ball, giving time to grab it. The robot starts to slow down earlier if it faces toward its own goal than if it faces toward its target goal. The difference is due to the fact that rUNSWift didn't want the robot to hit the ball toward the own goal accidentally.

Stealth dog and velocity prediction were used in the last year hover to ball. If the stealth dog conditions are triggered, then the robot would stealth around an opponent. If the velocity prediction is activated, then the robot would move towards a particular heading.

As stated in the 2003 report (page 223 of [13]), the fact that maximum forward and maximum left component are based on the turn component is not making much sense. This problem has been fixed in this year's hover to ball.

5.8.2: Hover to ball before the Australian Open

Hover to ball was rewritten for the Australian Open. This rewrite is necessary because canter and offset walk are not used for the ERS-7 robots.

Due to fact that velocity prediction is poor, it is removed in the new hover to ball. The new algorithm is simpler than the previous version since there is no need to change the walk type. Before the Australian Open, only low gain normal walk was developed

successfully. The goal of the rewrite is to minimise robot crashes due to battery overcurrent. As mentioned in section 4.2.4, allowing the turn and left components both be nonzero would lead to battery overcurrent more easily than if either of these components is zero. The new hover to ball algorithm ensures at least one component — left or turn must be zero.

Decisions:

- If the ball heading is small and the ball distance is large, then both turn and forward component are used. The forward component is equal to the maximum forward speed, the turn component is related to the ball heading similarly as the old algorithm . The long distance should provides enough time for the robot to turn while it walks at its fastest speed, in particular normal walk is a slow walk type.

- Otherwise if the ball heading is large, only the turn component would be used. The robot turns toward the ball until the ball heading is not large compared to the ball distance.

- If the above conditions are not satisfied, then the ball distance and ball heading are assumed small. In this case the forward value is set to the maximum forward speed and the turn value is set to a small value for small turning.

Notice that none of the decision allows left and turn both become nonzero. In the first and third decision, left is zero, the robot turns toward the ball while walking at its maximum forward speed. In the second decision, the robot turns toward the ball on the spot until the ball heading is small. The decisions are not based on the turning direction even the ERS-7 robot's body is not symmetry. This is because the low gain normal walk is slow and hence stable enough to turn without slipping.

For the competition, the first decision was satisfied when the absolute value of the ball heading is smaller than 22 degrees and the ball distance is greater than 60cm. The second decision was satisfied when the first decision failed and the ball heading is greater than 18 degrees. These values were hand-calibrated.

The new algorithm also ensures the robot to slow down for ball grabbing when it is close the ball.

From experiments, the new algorithm allowes the ERS-7 robots survive longer than originally. Before this change, no single robot could survive for a 10 minute game. After this change and other changes were applied, a single robot could survive for a 10 minute game without crashing. However the robots were slow — the left component was not used at all. This is not a significant problem since the robots can not walk fast anyway with the low gain normal walk.

5.8.3: Hover to ball after the Australian Open

The hover to ball was rewritten again after the Australian Open. This time two walk types are used — elliptical and normal walk. Elliptical walk is used when the robot wants to walk straight quickly. Recall from section 3.3.2, elliptical walk can walk forward quickly, but it cannot turn in particular the robot cannot stop with the elliptical walk. Normal walk is used when the robot wants to turn smoothly. Normal walk is slow, but it allows good turning and the robot can stop with the normal walk. The walk types are switched dynamically during a game.

Similarly as the previous year, the new hover to ball algorithm makes decision based on the ball heading and the distance to the ball. The idea is to turn toward the ball with normal walk and then walk straight with elliptical walk. Both walks must be used — if only the normal walk is used then the robot would be too slow, if only the elliptical walk is used then the robot cannot turn reliably.

Decisions:

- If the ball heading is large, then normal walk would be used for quick turning on the spot. Turning in the direction of the ball.
- If the robot wants to slow down (eg: ball grabbing), then normal walk would be used because normal walk allows the robot to control the forward speed.
- Otherwise the ball distance is large and the ball heading is small. The robot walks with elliptical walk. In this case the robot has enough time to complete the turn even with the elliptical walk.

Notice that unlike the previous year algorithm, the maximum forward, left and turn values are not set. This is not necessary because the new method basically draws a small circle around the ball. The robot turns toward that circle, walks with elliptical walk toward that circle until it reaches the circle. After the

circle is reached, the robot uses the normal walk to slow down and gain possession of the ball. In the event of the elliptical walk, the forward speed is always constant, the left speed is always zero, the turn speed is set to the halve of the ball heading with maximum 30 degrees. In the event of the normal walk, if it is used for turning on the spot, the forward and left are set to zero. The turning component is set to the halve of the ball heading with maximum 30 degrees.

Previously hover to ball was only used by the attacker. Supporter and striker (see section 5.13) did not use the hover to ball, they walked toward their desired position with the slow canter walk. This year all the robots including the supporter and striker uses the hover to ball technique, speeding up their movement.

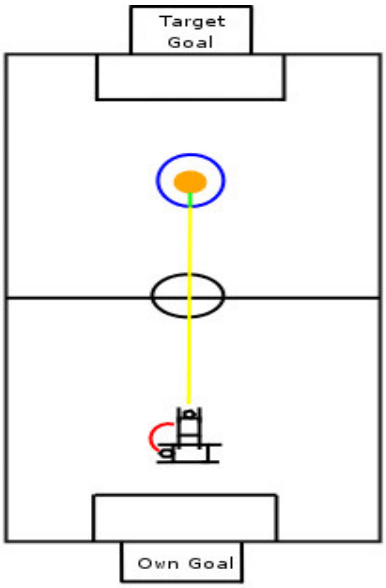


Figure 5.20: Hover to ball. The robots turn toward the ball with the normal walk (red line). Walk straight toward the ball with the elliptical walk (yellow line). Finally after the robot reaches the small circle drawn around the ball, the robot uses the normal walk to slow down (green line).

```
/*
   Copyright 2004 The University of New South Wales (UNSW) and
   National ICT Australia (NICTA). This file is part of the 2004
   team rUNSWift RoboCup entry. You may redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of
   the License, or (at your option) any later version as modified
   below.  As the original licensors, we add the followin
   conditions to that license: In paragraph 2.b), the phrase
   "distribute or publish" should be interpreted to include entry
   into a competition, and hence the source of any derived work
   entered into a competition must be made available to all parties
   involved in that competition under the terms of this license.
   In addition, if the authors of a derived work publish any
   conference proceedings, journal articles or other academic
   papers describing that derived work, then appropriate academic
   citations to the original work must be included in that
   publication. This rUNSWift source is distributed in the hope
   that it will be useful, but WITHOUT ANY WARRANTY; without even
   the implied warranty of MERCHANTABILITY or FITNESS FOR A
   PARTICULAR PURPOSE.  See the GNU General Public License for more
   details. You should have received a copy of the GNU General
   Public License along with this source code; if not, write to the
   Free Software Foundation, Inc., 59 Temple Place, Suite 330,
   Boston, MA  02111-1307  USA
*/


# Move to specified position given the global coordinative.
def saGoToTargetFacingHeading(targetX, targetY, targetH, maxSpeed =
                              7, maxTurn = 30):

    # ariables relative to self localisation.
    selfX     = Global.selfLoc.getX()
    selfY     = Global.selfLoc.getY()
    selfH     = Global.selfLoc.getHeading()
    relX      = targetX - selfX
    relY      = targetY - selfY
    relH      = HelpShort.normalizeAngle_180(targetH - selfH)
    relX     += Constant.DOG_LENGTH/2.0/10.0*(math.cos(math.radians
                (selfH)) - math.cos(math.radians(targetH)))
    relY     += Constant.DOG_LENGTH/2.0/10.0*(math.sin(math.radians
                (selfH)) - math.sin(math.radians(targetH)))
    relD      = HelpShort.getLength((relX, relY))
    distance  = HelpShort.getDistanceBetween(targetX, targetY,
                selfX, selfY)
    inCircle  = distance <= 40
    faceH     = HelpShort.getHeadingToFaceAt(targetX, targetY)

   # Inside the circle?
    if not inCircle:
            if abs(faceH) >= 30:
                  HelpLong.setNormalWalk(0, 0, HelpShort.CLIP(faceH,
                                         maxTurn))
            else:
                  HelpLong.setEllipticalWalk(7, 0, HelpShort.CLIP
                                        (faceH/1.5, maxTurn))
    else:
            relTheta = HelpShort.normalizeAngle_180(HelpShort.
                        RAD2DEG(math.atan2(relY, relX)) - selfH)

            forward  = HelpShort.CLIP(relD, maxSpeed) * math.cos
                        (HelpShort.DEG2RAD(relTheta))

            left = HelpShort.CLIP(relD, maxSpeed) * math.sin
```

```
                    (HelpShort.DEG2RAD(relTheta))

        turnCCW  = HelpShort.CLIP(relH, maxTurn)
        HelpLong.setNormalWalk(forward, left, turnCCW)
```

Figure 5.21: Hovertoball skill allowing the robot to move to a particular place on the field.

5.8.4: Performance

The normal and elliptical walk combination works very well together. The robots are able to walk quickly while turning smoothly.

From the observations in the world open, rUNSWift robot's turning speed is too slow to being competitive. The future teams should improve the turning speed and integrate it within the hover to ball.

5.9: Visual opponent avoidance kick

5.9.1: Introduction

Visual opponent avoidance kick (VOAK) allows a robot to shoot through the largest gap between the opponent goalie and the target goal. It is an useful skill when the robot is close to the target goal and want to shoot accurately, preventing the opponent goalie to block the shoot. To do this the robot must grab the ball and turn it until it faces the largest gap.

VOAK can shoot the ball very accurately but it is also slow. Few seconds are needed for the robot to turn and align itself with the gap, these few seconds is enough for the opponent robots to block the VOAK. In general VOAK is only useful when no opponent robots are nearby.

5.9.2: 2003 VOAK

At the start of the VOAK, a three second timer starts. If the

timer reaches zero, the robot immediately aborts the VOAK because of the three second ball holding rule.

The first step of the VOAK is to grab the ball. To stop the ball a robot must grab it for a sufficient number of frames. The ERS-210 robots are able to grab the ball with their paws. When the robot grabs the ball it must move its head down for ball tracking.

The next step is move the head up while holding the ball. If the robot sees a ball, it immediately aborts the VOAK and chase the ball since there is nothing to shoot. If the target goal is not seen the robot turns the ball toward the GPS goal until the robot sees the target goal or the three seconds timer expires.

In the event that a target goal is seen, the robot computes all the gaps exist between the opponent robot and target goal. The robot determine the largest gap by the number of target goal colored pixels that are present in the lower half of each gap. The width of the gaps is not used because it can be misleading (page 218 of [13]).

The robot turns itself and the ball until it is lined up with the largest gap. When this occur the robot perform a front kick.

To avoid switching between multiple comparable size gaps, once the robot believes it has selected the largest gap, it turns itself toward that gap, ignoring the other gaps.

For more details, refer to the last year's thesis [13].

5.9.3: 2004 VOAK

A new VOAK has been developed this year for the ERS-7 robots. The new VOAK is very similar as the old VOAK, only the physical actions are changed. Gap computation remain unchanged since it is robot independent.

The old VOAK cannot be used for the ERS-7 because:

- ERS7 cannot grab the ball with its paws (section 4.3).
- ERS7 must use its head to hold the ball while turning. Also it cannot see the ball while turning (section 4.3).

- Front kick does not work for the ERS-7. (section 3.5.1).

The first step at the start of the new VOAK is to grab and hold the ball with the head. The ball must be grabbed for few frames to prevent the ball roll out. The robot then looks up, expecting to see the target goal. If the ball is seen, the robot aborts the VOAK immediately because the ball has rolled out. If the target goal is not seen the robot moves its head down and turn the ball toward the GPS target goal. When it believes it is facing toward the GPS target goal, it moves its head up again. The robot aborts the VOAK if it still cannot see the target goal.

In the event that the robot can see the target goal, it determines the largest gap similarly as the old method. Once the gap has been determined, the robot sticks with it and ignore other gaps. If the robot finds itself already align with the gap, it shoots the ball immediately. Otherwise the robot moves its head down, hold the ball with its paws and head and turn toward the gap.

Since the robot cannot see the target goal while it is turning. It calculates the number of frames required for turning before it starts to turn, and stop the turn when the counter reaches zero. After the robot completes the turn, it moves its head up and expecting to see the target goal. If it cannot see the target goal, then it aborts the VOAK and chase the ball. Turning again is not recommended because the three seconds timer would be likely to expire.

In the event that the robot can see the target goal after the turn, it kicks the ball with hand kick (section 3.5.4). Left hand kick often hit the ball to the right around 30 degrees. Hence if the robot wants to do a left hand kick it would stop 30 degrees left from the center of the gap. Similarly for the right hand kick. Left hand kick is performed if the robot turns clockwise otherwise it would do a right hand kick.

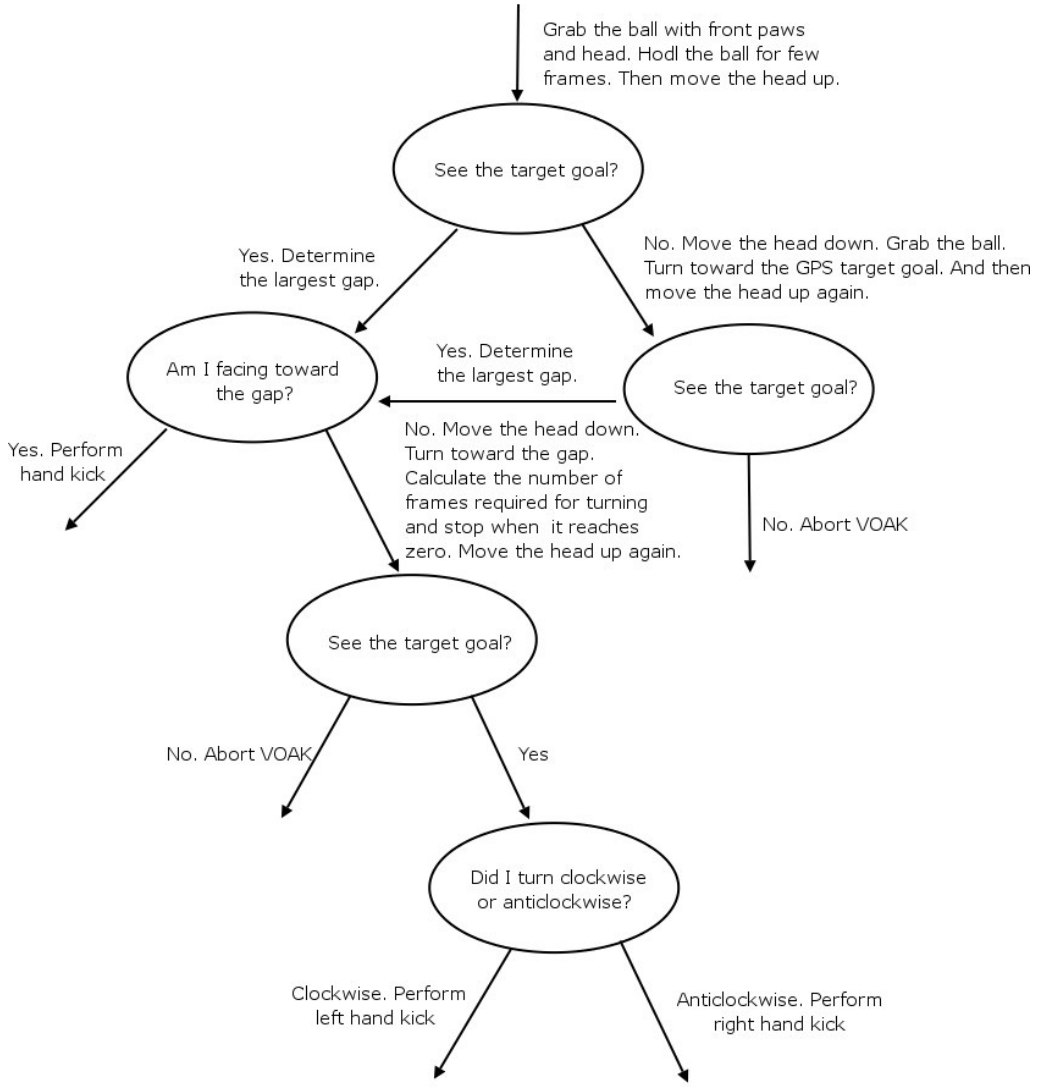


Figure 5.22: VOAK flow diagram.

5.9.4: Performance

The new VOAK cannot perform as well as the old VOAK because:

- Poor vision. Robot recognition and goal recognition are poor specially for the blue robots and blue goal. Often the red robots cannot recognized the blue opponent goalie and blue target goal. Only the blue robots can perform the VOAK because red color is more easily to be recognized [5] and [8].

- The robot must moves its head down while turning. Hence it cannot see the target goal. Although the number of frames required for turning is calculated before the robot turns, it may not be

accurate in particular if the opponents push our robot.

VOAK was not used in the competition since it was invented two days before the world open and hence not enough time to integrate it in the attacker's strategies.

5.9.5: Future Development

Vision has a significant impact on the VOAK performance. Reliable VOAK can only be developed with a decent goal and robot recognition.

The future generation should definitely try University of Technology of Sydney (UTS) VOAK. In the world open, UTS robots grabbed the ball with their mouth so that they could see the target goal while holding the ball. Their robots move sideway with the ball.

5.10: Bird of Prey

5.10.1: Overview

The Bird of Prey (BOP) algorithm was invented last year. It is a dynamical role assignation algorithm. It assigns a forward as a bird. The bird would place itself between the own goal and the ball.

BOP is a very important skill. Unlike some of the other teams, rUNSWift does not have a defender. Most of the time one forward is assigned as an attacker, another forward is assigned as a supporter and the third robot stayed slightly behind its teammates. If the ball rolls pass the forwards, they need to get back for defense as quickly as possible. Sometimes it is no good to walk directly to the ball since the robot would face toward the own goal when it reaches the ball. BOP allows the robots to take a curve locus, positioning itself between the own goal and the ball and possibly preventing opponent robots to push the ball toward the own goal

BOP would only be activated when cover defense is required. Cover defense is considered necessary “*when none of the forwards in an area enclosed by 150 degrees, centered straight down the*

defensive direction, 10 cm in front of the ball. The angle allows for the fact that a teammate across the opposite side of the field than the ball, but level with it, is not in an adequate position to counter an enemy offensive maneuver ” (page 179 of [13]). If a forward is in that area then it would probably be able to walk directly toward the ball and face toward the target side.

Once the bird is activated it would be deactivated if a forward is in the area. Hysteresis is necessary otherwise the robot would switch between activating and deactivating BOP. For deactivation, the angle is reduced to 90 degrees and the distance to zero. See figure 5.23.

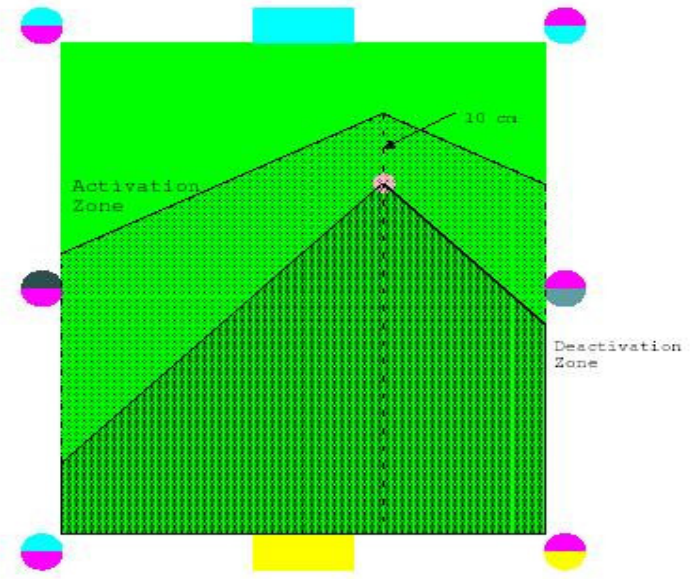


Figure 5.23: BOP regions for checking for defending forwards. (Image courtesy of 2003 UNSW RoboCup report, page 178 of [13])

In 2003, only one bird was allowed (page 178 of [13]). The bird would only be effective if the bird has enough room to take a curve locus. Hence the bird is chosen as the forward with the greatest x-distance from the ball.

The bird maintains the ball at a particular heading (45 degrees in 2003) until it is deactivated. This form a curve locus around the ball. Walking in a curve allowing the robot to place itself between the own goal and the ball. A line is drawn from the ball to the center of the goal, when the bird is on the left side

of this line then the locus would curve around the right side. Similarly when the bird is on the right side of the line. A circle with radius around 18cm is drawn around the ball. The bird turn relatively slow when it is outside the circle. When it reaches the circle, it slowes down and turn quickly toward the ball.

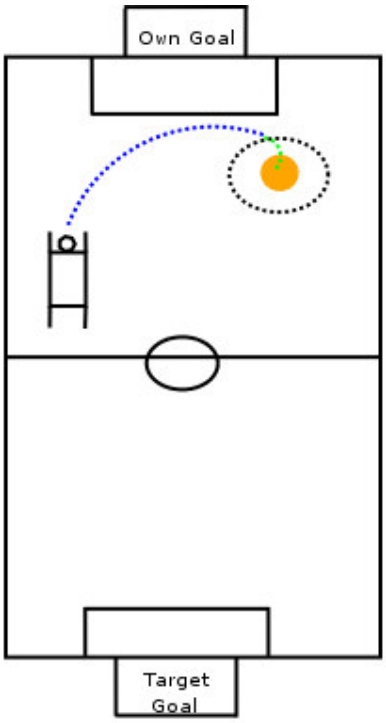


Figure 5.24: Bird takes a curve locus toward the ball. Positioning itself between the own goal and the ball. When the robot reaches a small circle drawn near the ball, it start to turn quickly toward the direction to the ball.

The curved locus often leads to the robot entering into the defensive goalbox and hence taken off the field for 30 seconds. When the bird detects it is going to enter into the defensive goalbox, it would change its locus and walk on the goalbox line, see figure 5.25.

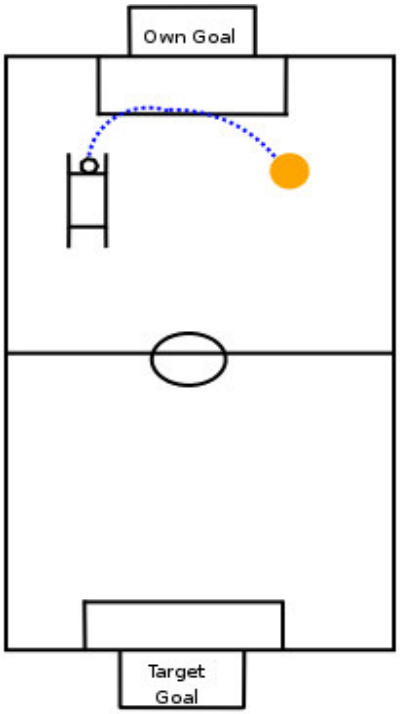


Figure 5.25: Bird avoiding goal line.

5.10.2: 2004 Bird of Prey

This year the BOP has been ported into Python. The implementation is similar as the previous year. Previously the bird turned with canter walk when it reached the circle drawn around ball. Otherwise zoidal walk would be used. This year elliptical walk has replaced the zoidal walk, while the normal walk replaces the canter walk.

Previously only one bird was allowed. This year all the forwards would becoming bird. The idea is instead of walking directly toward the ball and face toward the own goal, all the forwards should be defensive and take a curve locus. The birds are in better position to counterattack after they gain the possession of the ball. Sometimes the robots would be called pushing if they take a direct path to the ball, pushing the opponent robots on the way.

5.10.3: Conclusion

BOP with multiple birds worked extremely well in the competition. It allowed the rUNSWift robots to attack aggressively and defense quickly. The birds were often able to block the opponent attack.

In this year's competition, some of the teams walked straight toward the ball, grabbed it and turned it toward the target side. This strategy is faster than BOP. The future rUNSWift teams may consider to assign two robots as the bird and assign the robot closest to the ball to take a direct path toward the ball for ball grabbing.

5.11: Goalie

5.11.2: 2003 goalie

Goalie is the only robot allowing to stay in the goal box. Its job is to prevent the enemy to score and clear the ball.

Goalie's positioning is very important. A good goalie should stand between the ball and its own goal, effectively block enemy shoot. In 2003, if the goalie could see the ball, it would placed itself on the intercept between the line through ball and center of the goal, and a semi-circle centered at the center of the goal. See figure 5.26.

However if the robot could not the see the ball, the goalie positions itself further behind by using an ellipse instead of the semi-circle. The idea is that if the ball is visible, the goalie should be aggressive and place itself further forward to reduce the size of gaps the enemy opponent can shoot.

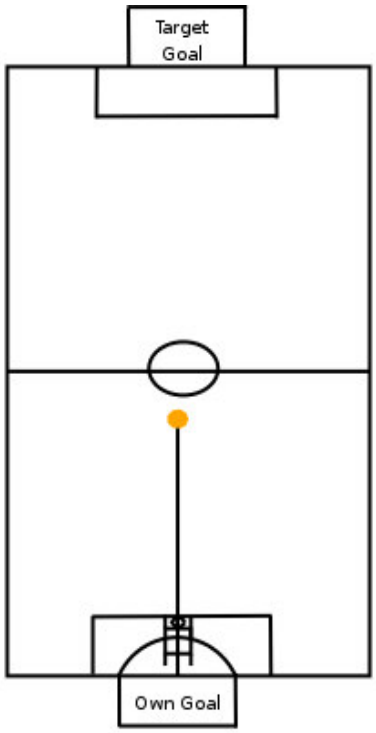


Figure 5.26: Goalie's positioning when it sees the ball.

If the ball is not visible either gps or wireless ball would be used. In either case the goalie would be more defensive and placed itself near the goal line. Since the middle and far beacons could not be seen very well, the goalie should be positioned in front of the goal line so that it could see the two close beacons.

The previous year goalie would spread its arms to stop a fast moving ball, known as goal block. Since the goal block has a slow recovery time, it should not be activated too frequently. It was activated when all the following conditions were satisfied:

- The ball was close to the goalie or close to the goal line. The goalie should not do a goal block if the ball is far away.
- The goalie believed it could block the ball with its arm spread. Velocity prediction was used and it must be reliable.
- The ball traveled fast enough to be able to collide with the goalie within the next second. This condition ensured the goalie to block at the right time.

An important decision a goalie must make – attack or defense,

If the goalie decides to attack, enemy robots may shoot the ball through gaps. If the goalie decides to defense, the enemy robots may have enough time to regroup and attack the ball. The last year goalie would attack the ball if:

- The ball was close to the goalbox (10cm within the goalbox). So that the goalie would stayed inside the goal box.
- The distance between the goalie and the ball was small and no opponent robot was close to the ball. This condition ensured the goalie cleared the ball before the opponents gained the control. The further the goalie was from the goal line, the closer the ball had to be for the ball distance was considered as small. These conditions ensured the goalie attacked when it was safe to do so.

If the goalie decided not to attack, it stayed in front of the goal line defensively, active localise and patiently searched for the ball.

If the ball was under the goalie's chin, the goalie turned toward the gap and kicked the ball with the lightning kick. If the the goalie could not see a gap, the goalie performed a chest push and then executed the algorithm again to decide what to do next.

For more details, refer to last year's thesis [13].

5.11.2: 2004 Goalie

The last year goalie performed well in the 2003 competition. However the codes are messy and hard to maintain. Also the lightning kick cannot be used this year, since the offset between the robot's front paws and its chest is too small.

This year after the Australian Open, the team decided to rewrite the goalie in Python. The new goalie offers some improvements but also some drawbacks.

The new goalie places itself further behind than the old goalie. It positions itself on the goal line. In general unless the goalie wants to attack, it should positions itself on the goal line to reduce the size of gaps through which opponent robots can aim. The old ERS-210 goalie must position itself in front of the goal

line, otherwise it can't see the close beacons. The middle and far beacons are too small and give little information. Although the new ERS-7 robots also cannot see the far beacons very well (middle beacons have been removed), they can use the edge detection to localise by looking at the goal lines. Refer to Whaite.D's thesis for more details [7].

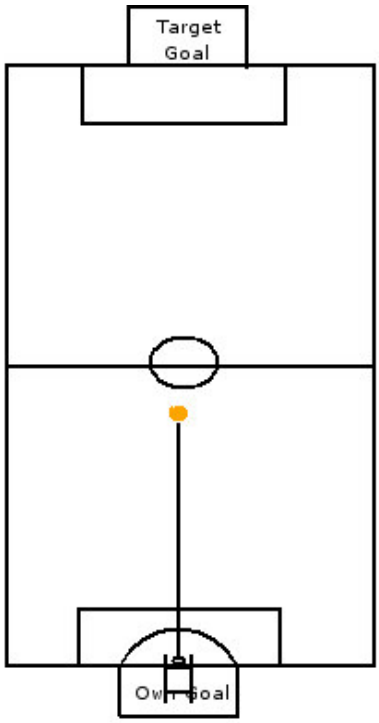


Figure 5.27: Goalie positions istelf on the goal line.

The conditions to trigger when to attack the ball and when to defense are very similar as the previous year and hence not repeated here.

However the actions taken when the goalie has gained possession of the ball is different, which is related to how the goalie approaches the ball. If the distance to the ball is far, then paw kick would be used. Paw kick is preferred because it is fast — the robot doesn't need to slow down while approaching. The long distance gives enough time for the goalie to align its paw with the ball. Paw kick would not be useful if the distance to the ball is small because the goalie doesn't have time to align its paw with the ball. In the event of the ball distance is small, hover to ball would be used. Hover to ball allows the robot to move toward the ball directly.

If hover to ball is used and after the goalie has gained possession of the ball, it kicks the ball with either the side way kick or dive kick. Decisions:

- If the goalie's global heading is greater than 225 degrees or smaller than 315 degrees, the goalie is facing toward its own goal. In this case the goalie will not kick the ball to avoid own goal.

- If the goalie's global heading is smaller than 225 degrees and greater than 135 degrees, the goalie do a left sideway kick to hit the ball toward the right-hand side. Right sideway kick is not recommended because the ball may roll toward the own goal.

- If the goalie's global heading is larger than 315 degrees and smaller than 45 degrees, the goalie do a right sideway kick to hit the ball toward the left-hand side.

- Otherwise the goalie wants to kick the ball forward, so it performs a dive kick.

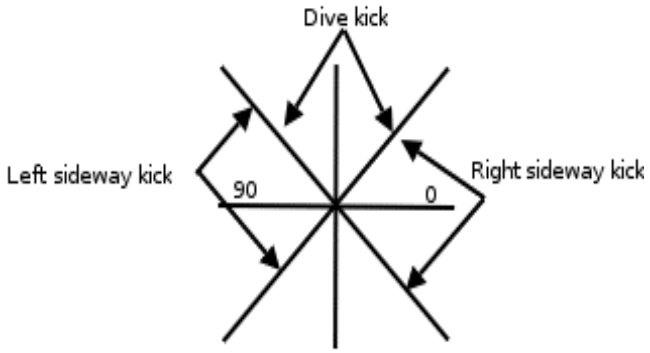


Figure 5.28: Goalie's kicking decision based on global heading.

Notice that the new goalie cannot shoot through gaps, this is a major drawback. It is not implemented and ported because there is not enough time to do it before the competition. Furthermore a reliable gap detection algorithm cannot be implemented because of the poor robot recognition [5], in particular a red goalie cannot recognized the blue opponents. The new goalie also cannot spread its arm to block a fast moving ball.

The new goalie's decision tree has been redesigned. The new decision tree is flexible and maintainable. Previously the goalie kicked the ball immediately when the ball was under its chin, which kick it used was depend on the gaps and world model. As mentioned in section 5.3, this approach is not always the best. The new decision tree removes the restrictions and ask questions such as "Is sideway kick ok? If it is not ok, is it ok for me to perform dive kick?". Refer to section 5.3 for more details.

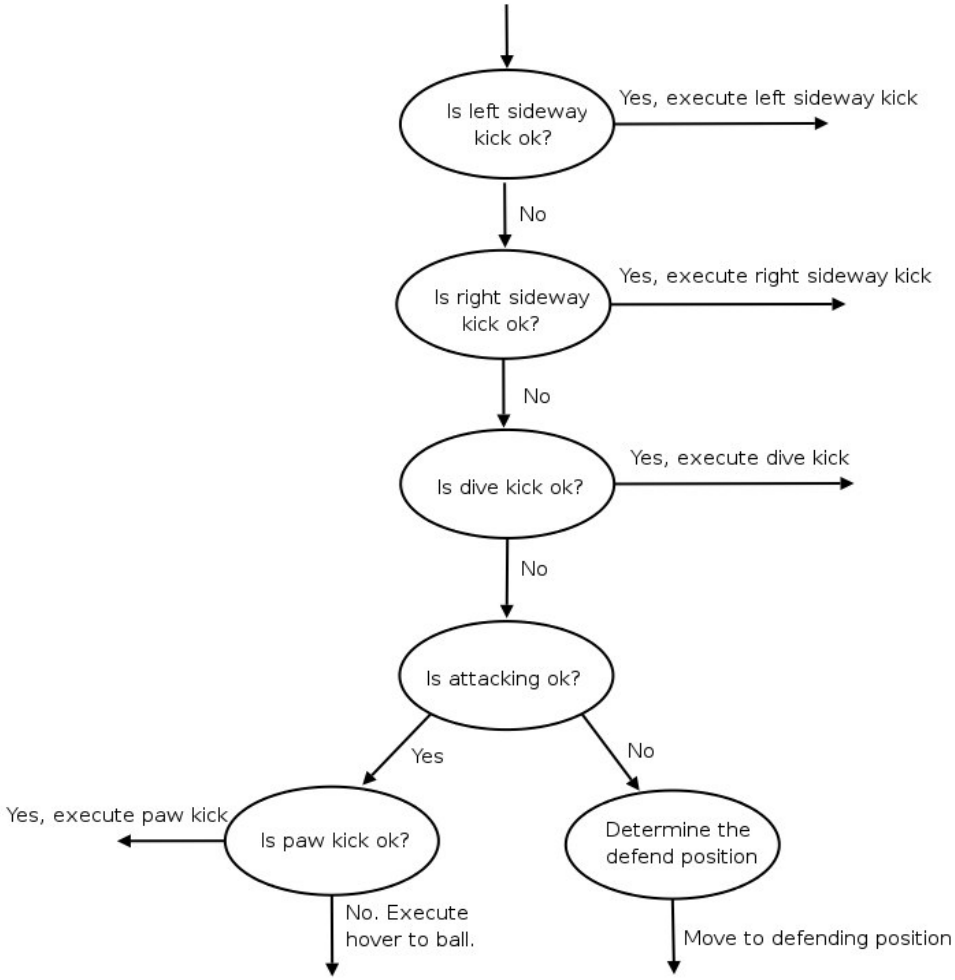


Figure 5.29: 2004 goalie's decision tree.

5.11.3: Conclusion and discussion

The new goalie positions itself better than the old goalie — it is more defensive than the previous goalie.

The new goalie can kicks the ball quicker since paw kick and sideway kick are fast. Previously the goalie turned toward the gaps and then shoot, which is slow. The ERS-7 can blocks better than ERS-210 simply because ERS-7 is larger in size.

However sometimes the new goalie performs poorly. Since it is unable to kick the ball through gaps, it cannot kick the ball past the opponent robots. In fact this is one of the reason rUNSWift got beaten by the Germany team in this year's competition. The goalie missed some golden opportunities to clear the ball through gaps.

The goal blocking and shooting through gaps are something future rUNSWift teams will want to add. The ERS7 goalie needs to redesign and improve in the future.

5.12: Ready Player

5.12.1: Game state

Each soccer match is composed of game states. Game state defines the actions allowed. Fail to comply may impose penalty.

This year a new game state known as *ready state* has been introduced. In this state all the robots are expected to move to their kickoff position. All other game states remain unchanged from the previous year.

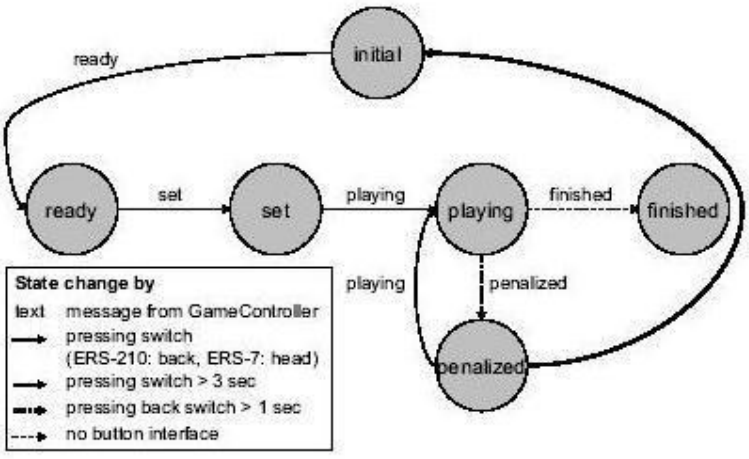


Figure 5.30: Game states flow diagram. (Image courtesy of Sony Four-Legged Robot Football League Rule Book 2004 [3])

A description of the game states.

- Initial state

  The robots are not allowed to move, they can't do anything but wait for the ready state.

- Ready state

  In this state, the robots walk to their kickoff position as quickly as possible. This state remains until all the robots reach legal position or the referee decides to switch into the set state.

- Set state

  In this state, the robots stop and wait for kick-off. If they are not at legal positions, they are manually placed . The robots are allowed to move their head and tail. Useful for localising before the game begins.

- Playing state

  In this state, the robots play a soccer match.

- Penalized state

  This state is reached when a robot is penalized for whatever reason, eg: pushing and ball holding. The robot is taken away from the field for 30 seconds.

- Finished state

  This state is reached when a half or the whole game is finished.

Either the gamecontroller or manually button pressing trigger the switching of game state. Refer to [3] for more details.

5.12.2: Ready state

Previously the robots needed to placed manually before a game started, this process is tedious. A team of intelligent soccer playing agents should know how to reposition themselves. A new state known as ready state has been introduced into the competition this year. This state occurs after the initial state or after a team has scored (ie. After playing state). In this state, all the robots need to walk to their kickoff position. This year rUNSWift has invented a ready player for the ready state.

If a robot can't move to a legal position then it is manually placed. So implementing a ready player is not compulsory, manual placement is always an alternative. However manual placement should be avoided because the robot has to be placed at least one robot length behind in contrast to if it moves autonomously. If the robots reposition themselves, "*Two field players of the attacking team can walk to positions between the center line and the middle of their side. They may put their leg on the center line, but no leg may be inside the circle line. The other field players (one of attacking team, three of defending team) have to be located behind the middle of their side ....., but have to stay outside their own penalty area with at least two feet.*" (page 10 of [3]). If the robots need to be manually placed, "*..... the kicking-off robot shall be one robot length away from the center circle, while one robot of the other team shall be just in front of one corner of the penalty area. The other robots shall be on the left and on the right of their own penalty area.*" (page 10 of [3]).

Since autonomously placed robots are allowed to position closer to the ball, there is an incentive for implementing a reliable ready player. See figure 5.31.

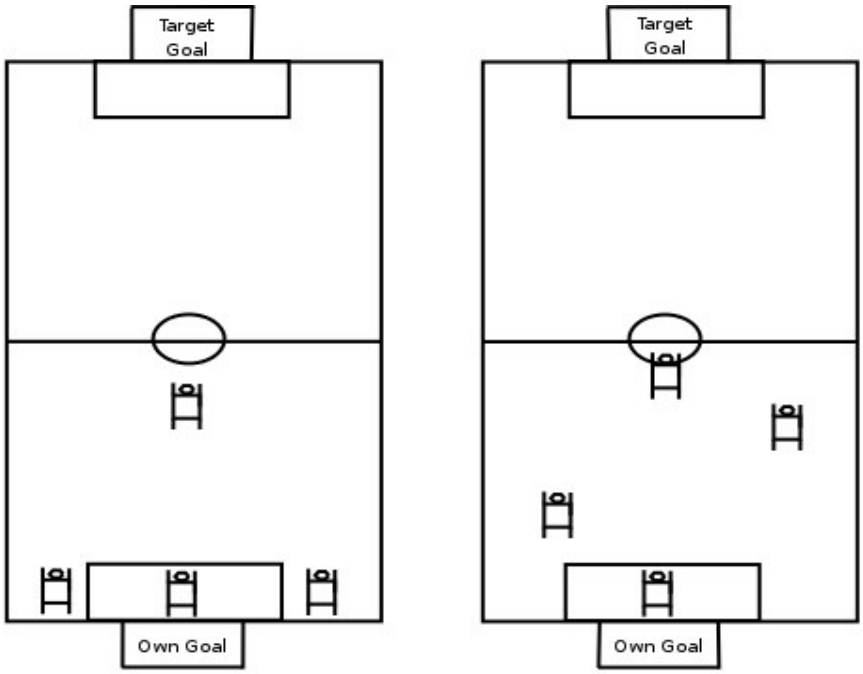


Figure 5.31: Manual placement and a possible autonomous placement for the attacking side. Notice that autonomous placement allows the robots to position closer to the center of the field.

Apparently the ready player is easy to implement. It only needs to walk to its kickoff position, no ball tracking is involved. However, in this year's competition, none of the team could implemented a perfect ready player. The challenges:

- Avoid teammate and opponent robots while repositioning. Avoiding opponent robots is a hard problem because you do not have any control over the opponent robots. In general this is the most the number one challenge for the ready player.

- Decide where to kickoff. The ready player should choose an easy reachable kickoff position, otherwise it may takes too long or never reach its kickoff position.

- The ready player needs to reach its kickoff position accurately. However in general this year's localisation is worse than the previous year, because two middles beacons are removed and ERS-7 has a poor camera.

- Decide how to reach the kickoff position. Sometimes walk straight toward the kickoff position may not be the fastest path.

The competition rule has not specified neither the minimum nor maximum time for the ready state. The ready state is finished when a referee believes the robots should switch into the set state. However it is a good idea for the ready player to reach its kickoff position as quickly as possible, so that it is well prepared when the game starts. Having said that, speed is not as critical as accuracy. A slow but accurate ready player often outperform a fast but inaccurate ready player.

Ready player was developed after the Australian Open. Before the Australian Open, due to battery overcurrent (section 4.2) ready player cannot be developed.

5.12.3: Legal and Ideal kickoff position

Legal kickoff position (LKP) is a position that is valid for kick-off. Goalie has a unique LKP. Forward's LKP is depend on whether the team is receiving or kicking-off. See figure 5.32, 5.33 and 5.34.

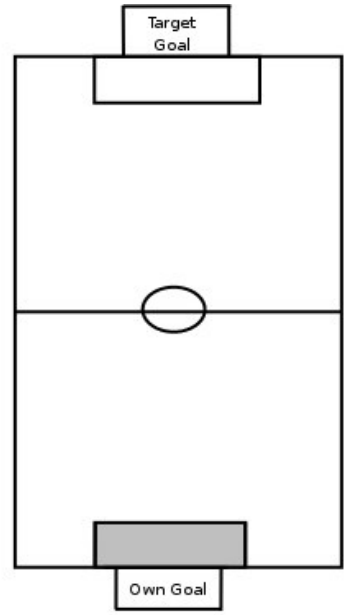


Figure 5.32: Goalie's legal kickoff position is shaded. Note that the goalie has to stay inside the penalty area with at least two feet.

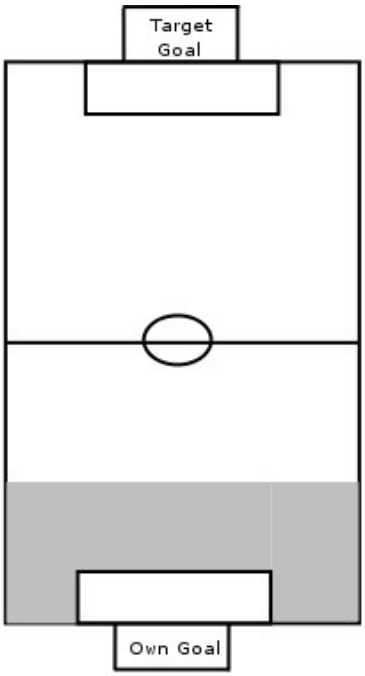


Figure 5.33: Forward's legal kickoff position is shaded. Assume the team is receiving.

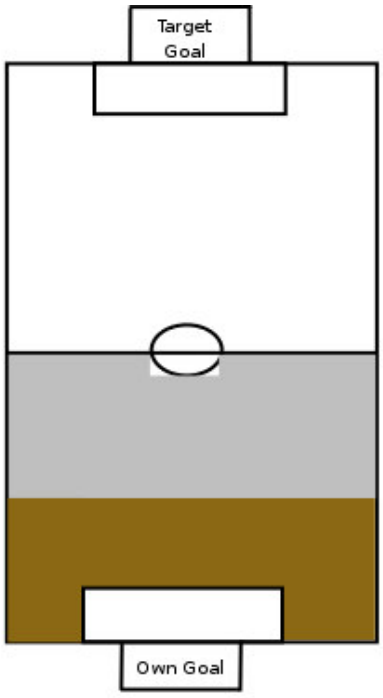


Figure 5.34: Forward's legal kickoff position is shaded. Assume the team is kicking-off. Note that only maximum two forwards can position within the grey area.

Unfortunately not all the LKPs are ideal for kicking off. Position that is ideal for kicking off is known as Ideal kickoff position (IKP). Note that IKP is a subset of LKP, but not the other way around. If the robots cannot reach a LKP, they must be manually placed. Even the robots are able to reach a LKP, sometimes they should still be manually placed. For example in figure 5.35, assume the team is kicking-off. One of the robot should be manually placed and position near the center of the field.

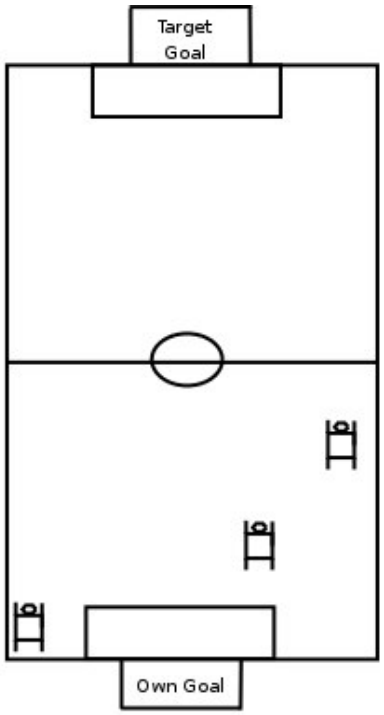


Figure 5.35: IKP and LKP. Assume the team is kicking-off. The robots are positioned correctly, manual placement is not compulsory. However the robots are not positioned well, one of the robots should position near the middle of the field.

There are infinite number of possible kickoff strategies. The kickoff strategy chosen by rUNSWift is simple: place the robots close to the ball (center of the field). One forward always positions itself in the center of the field and the other two forwards position themselves at the sides. The idea is to leave room between the robots, so that they wouldn't bunch up after the game begins. The robots positioning at the side shouldn't be position too close to the border. In any case the robots must face toward the target side.

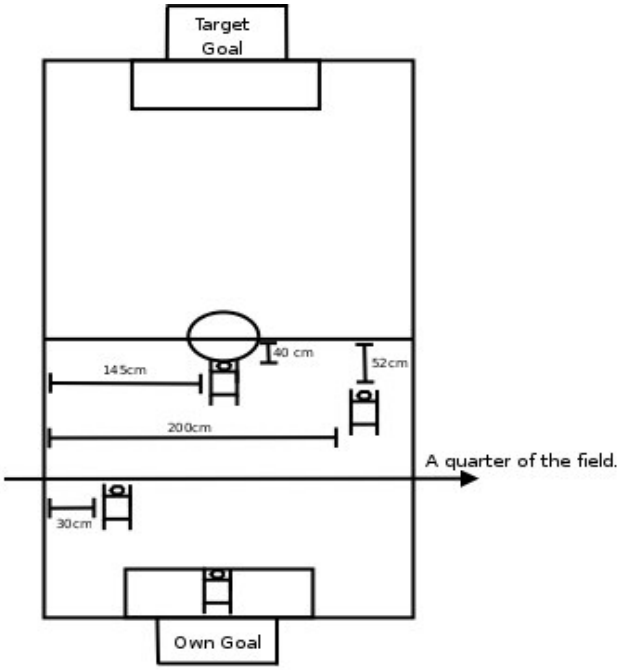


Figure 5.36: rUNSWift kickoff position during kick-off..

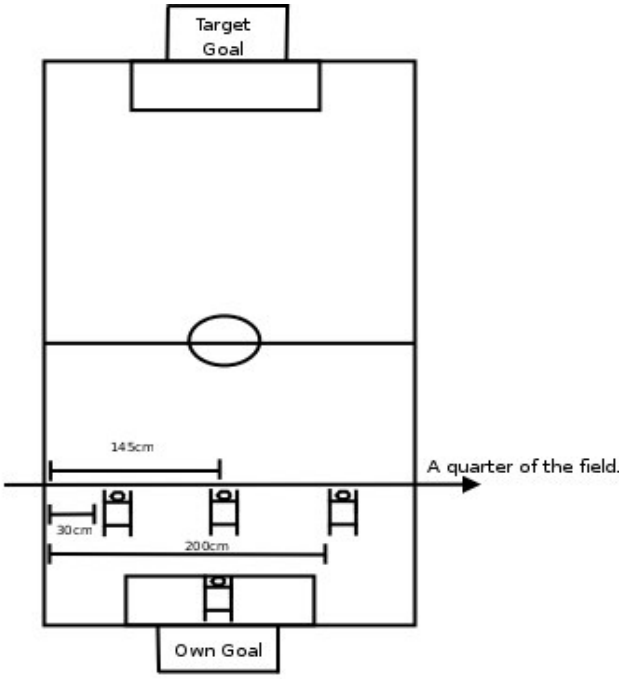


Figure 5.37: rUNSWift kickoff position during receive.

5.12.4: Ready player algorithm

The ready player can be broken down into four phases: localise, position assignment, walk and adjust. Each phase is independent of the others.

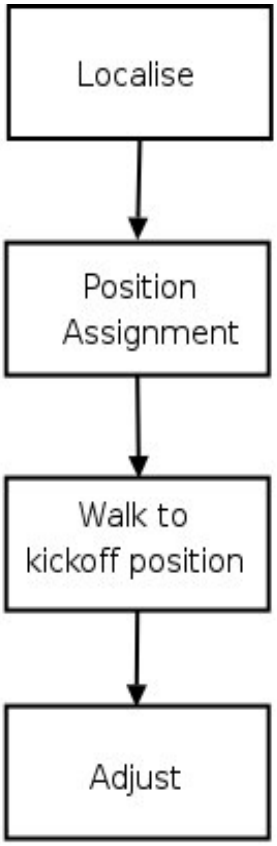


Figure 5.38: Block diagram of the ready player.

5.12.5: Phase 1 - Localise

Localise is the first phase. In this phase, the robots stop and pan their head for localise. This action is known as stationary localise. This phase is short and only lasted for few seconds.

Ready state arrives after a team has scored a point and often when this occur the robots are mislocalised, because they are concentrated on tracking the ball. For example dribbling requires

close-ball tracking, not allowing the robots to perform active localise. Also, the robots may need to push the opponent goalie for scoring, robot pushing may lead to mislocalisate because the motion update confuses the localisation module. Since the ready player heavily rely upon accurate localisation, stationary localise is necessary. Stationary localise helps the robot to see the field landmarks.

If the robot sees the target goal and its distance is small (eg: inside the target goal), then it turns toward the own side while localising, since the robot needs to turn anyway during the walk phase (see section 5.12.10). Most importantly the robot may not see any beacon while it is inside the target goal. Similarly apply when the robot sees the own goal and its distance is small.

Always turn while localising was tried. Not much improvement was seen except for few seconds improvement since localising and turning were done simultaneously. Sometimes an incorrect motion update while turning affect the robot's localisation. In general accuracy is more important than speed for the ready player, always turn while localising is not used in this year's competition.

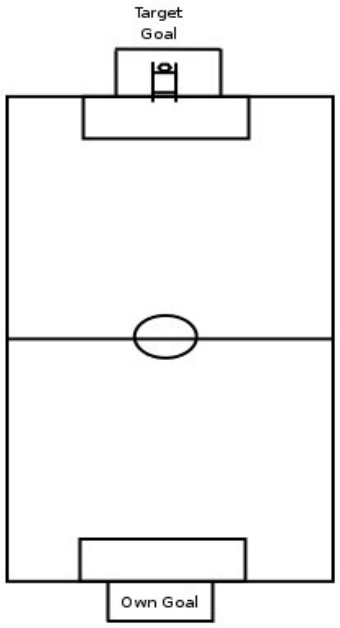


Figure 5.39: The ready player can't see any beacon with stationary localise. Hence it should turn while panning its head.

5.12.6: Position assignment

The next step after stationary localise is assigning kickoff positions. rUNSWift has a straightforward kickoff strategy – the kickoff positions are predefined. Unfortunately assigning kickoff positions is not straightforward. For example in figure 5.40, we have four robots: robot A, B, C and D. Robot D is the goalie.

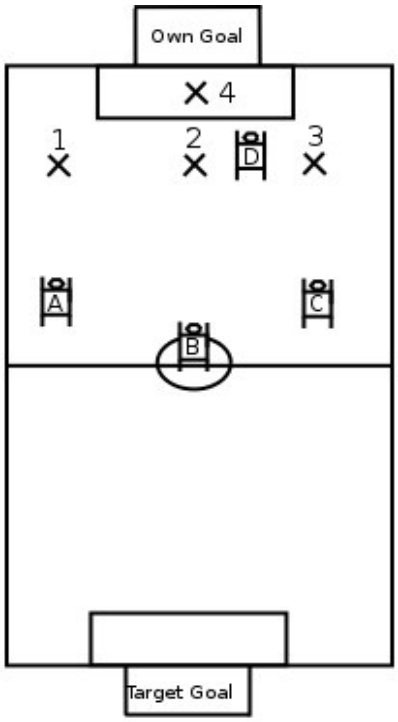


Figure 5.40: How to assign the kickoff positions?
For example where should robot A goes?
Robot D is the goalie.

The ready player often walks straight to its kickoff position, see section 5.12.9 for more details. A good position assignment algorithm should assigns positions such that the robots can walk straight most of the time, without crashing with each other. Two methods were tried this year: static kickoff position assignment and dynamic kickoff position assignment.

Static position assignment

The static position assignment algorithm is very simple. Each robot is assigned a fixed player number, the kickoff position is

based on the player number. Hence the kickoff position is fixed during the whole game.

Lets take an example. Assume the following player number assignment:

Robot A => player number 3
Robot B => player number 2
Robot C => player number 1
Robot D => player number 0

Figure 5.41: Player numbers for the robots. Correspond to figure 5.48.

Lets assume the static position algorithm assigns the kickoff positions as:

Player number 0 => Position 4
Player number 1 => Position 1
Player number 2 => Position 2
Player number 3 => Position 3

In our example, robot A is assigned to position 3, robot B is assigned to position 2, robot C is assigned to position 1 and robot D is assigned to position 4.

Since the robots walk straight toward their kickoff position (section 5.12.9), they reach their kickoff position like in figure 5.42.

Static position assignment algorithm is useless, so it is not used in the competition. Most of the time it assigns kickoff positions incorrectly since it doesn't take the possibility of robot collision into account. For example in figure 5.49, robot A walks to position 3, it may hit robot C whose is walking toward position 1 and robot B whose is walking toward position 2. Note that the walk path of robot A, B and C intersect.

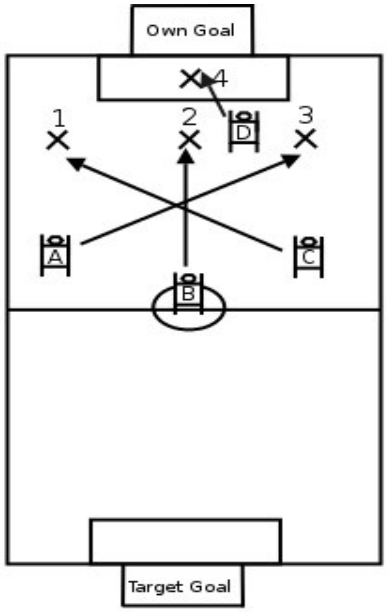


Figure 5.42: Static position assignment.

Static position assignment algorithm is also inefficient in terms of speed. For example in the above figure, clearly robot A should position itself at position 1 because it is the closest kickoff position, robot A needs to walk further in order to reach position 3.

Dynamic kickoff position assignment

Obviously the kickoff positions must be assigned dynamically. A dynamic kickoff position assignment (DKPA) algorithm was invented, it was used in this year's competition. The algorithm assigns kickoff position dynamically, depend on a number of factors.

Lets take an example, refer to figure 5.40. There are four robots: robot A, B, C and D, robot D is the goalie. For their player number. Robot A has the highest player number, robot B has the second highest, robot C has the third highest and robot D has the smallest player number. The robots have four kickoff positions to choose: position 1, 2, 3 and 4.

In DKPA, the forward with the highest player number and the

goalie always decide which kickoff position they want to go before any other robots. In our example, robot A is the forward with the highest player number and robot D is the goalie. Since there is only one legal kickoff position for the goalie, so robot D chooses position 4 for its kickoff position. Robot A has three choices: position 1, 2 and 3. How a robot decides a kickoff position would be explained in the next section, for now lets assume robot A picks position 1. After this decision is made, robot A broadcasts its decision (pick position 1) to its teammates through wireless. Robot B whose has the second highest player number then decide where to position itself. It knows position 1 is unavailable so it can only pick between position 2 and 3 (position 4 is for the goalie). Lets assume robot B picks position 2. Robot B broadcasts this decision to its teammates through the wireless. After robot C receives robot B's decision, it picks the only available kickoff position. Robot C knows position 1 and 2 are taken, so it can only pick position 3. Notice the pattern: robot A (highest player number) decides, then robot B (second highest player number) decides and finally robot C (the lowest player number) decides. This chain of decision is known as *ring algorithm*..

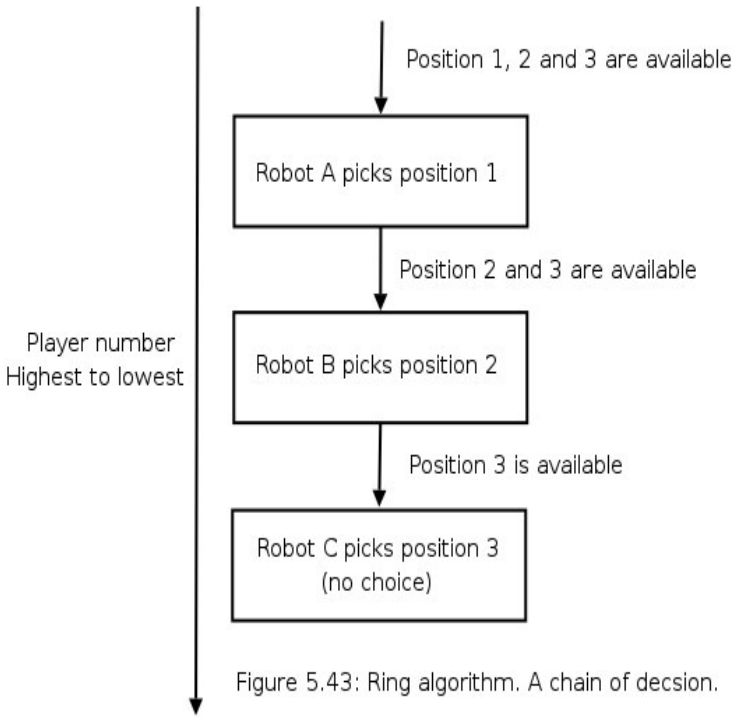


Figure 5.43: Ring algorithm. A chain of decsion.

In DKPA a higher player number forward can “steal” kickoff position from a lower player number forward. In our example robot A has the highest player number, it is allowed to change its kickoff position any time. Lets assume robot A suddenly believes position 2 is a better kickoff position (Of course position 1 is better, but robot D may be mislocalisated and think its nearest position is position 2). Robot A broadcasts this new decision to its teammates. Robot B knows its kickoff position (position 2) has been taken by robot A, so it must choose another position. Robot B has two choices: position 1 and 3. Lets assume robot B decides to “steal” position 3 from robot C. Robot B broadcasts this decision to robot C. Robot C knows position position 2 and 3 are taken, so it can only reposition itself at position 1. See figure 5.44.

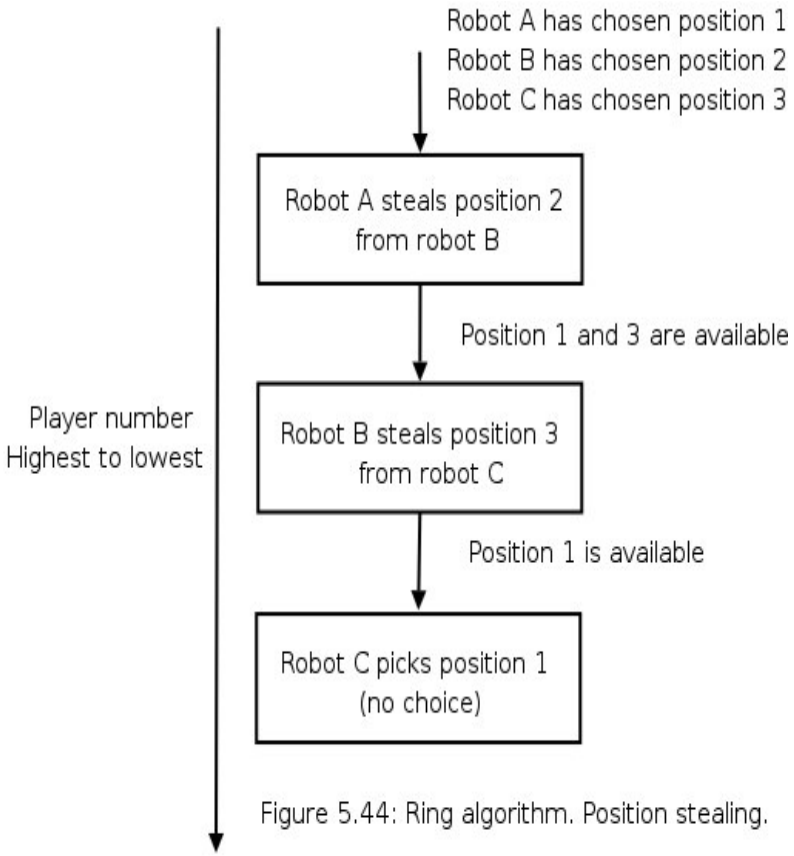


Figure 5.44: Ring algorithm. Position stealing.

So the forward with the highest player number is allowed to pick any kickoff position. It doesn't need to receive wireless information from any other robot because its decision process is independent of other robots. The second highest player number forward is allowed to steal kickoff position from the lowest player

number forward. Its decision process is depended on the highest player number forward. The lowest player number forward doesn't need to make a decision — only one available kickoff position. Goalie has only one possible kickoff position for it to reposition.

DKPA runs every frame until all the forwards believe they are positioned.

5.12.7: Ring algorithm performance

Ring algorithm is easy to understand and debug since the decision processes don't involve any synchronization. However there are some drawbacks:

- Convergence speed

  Only the forward with the highest player number is allowed pick any kickoff position, the overall convergence speed slower than a synchronous algorithm. If it mislocalisates, the overall position assignments may be incorrect since the lower player number forwards can't overwrite the highest player number forward's decision. Fortunately localisation is usually reliable during the ready state.

- Wireless problem

  DKPA depends on a reliable wireless communication. See figure 5.45. In the event wireless is broken or the number of teammates is not three (eg: a teammate crashes due to battery overcurrent), the ready player switches to the static kickoff position assignment algorithm.

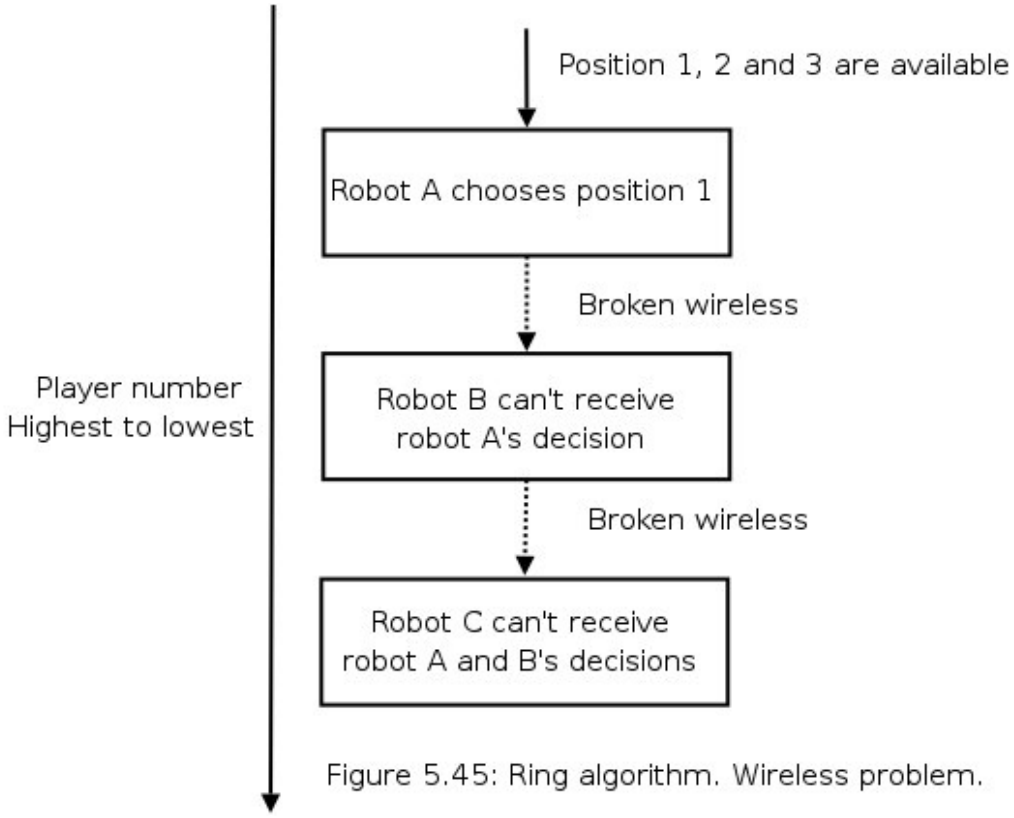


Figure 5.45: Ring algorithm. Wireless problem.

5.12.8: How to choose a kickoff position

Once again lets take figure 5.48 as our example. Robot C and D have only one kickoff position to choose. Robot A and B have multiple positions to pick.

Robot A (highest player number) chooses its kickoff position depend on its world model, in particular the position of the lower player number forwards — ie. robot B and C's global position. Decisions:

- If robot B and C are at the right-hand side of robot A, robot A would chooses the left kickoff position.
- If robot B and C are at the left-hand side of robot A, robot A would chooses the right kickoff position.
- Otherwise robot A chooses the middle kickoff position.

The idea is to avoid collision with the teammates, assuming the robots walk straight toward their kickoff position. Robot B chooses its kickoff position similarly. It has only two available kickoff positions, and it only need to check the robot C's global position.

- If robot C is at the left-hand side of robot B, then robot B would choose the right kickoff position.
- Otherwise robot C choose the left kickoff position.

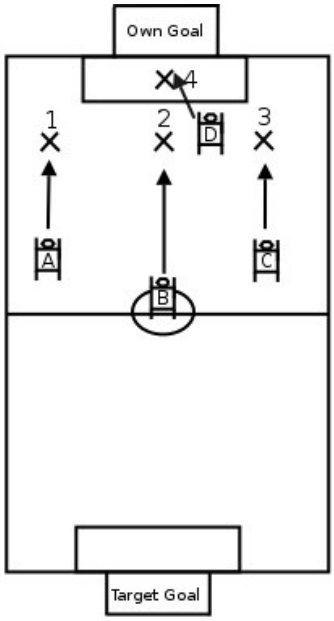


Figure 5.46: DKPA kickoff position assignment. Robot B and C are at the right-hand side of robot A, so robot A chooses position 1. Robot C is at the right-hand side of robot B, so robot B chooses position C. This leave position 3 for robot C. Robot D (goalie) always resposition itself at position 4.

5.12.9: Phase 3 — Walk strategy

This phase defines the actions required for repositioning. Two walk strategies were tried: walking backward and walking forward. In both case, the robots continually pan their head and localise while walking.

Walking backward

In this strategy, the ready player walks backward toward its kickoff position. This idea was taken from the German AIBO team, Germany Open 2004 [23]. The robots don't need to turn toward their kickoff position, they walk backward while facing the target side.

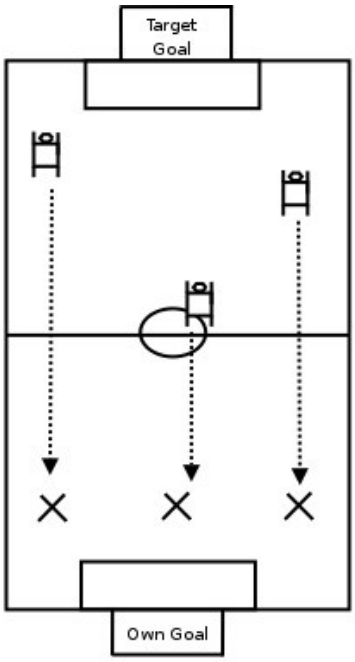


Figure 5.47: The robots walk backward toward their kickoff position.

Although this strategy is simple, it was not used in the competition. Opponent avoidance is impossible because the robots can't see their back. Also the robots walk slower when they walk backward.

Walking forward

This strategy was used in this year's competition. It is very similar as the hover to ball (see section 5.8), except for the fact that the ready players are walking to their kickoff positions. The robots turn on the spot with normal walk until they are facing toward their kickoff positions. And then, the robots walk straight with elliptical walk. After they have reached their destinations, they turn toward the target side with normal walk. See figure 5.48.

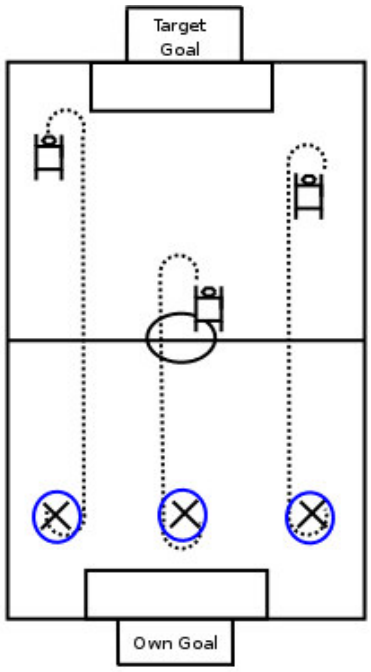


Figure 5.48: The robots turn and walk straight straight toward their destination with the elliptical walk. Once they have reached the small circle drawn around their kickoff position, they turn toward the target side slowly with the normal walk.

Advantages of this strategy:

- Fast, since the robots can walk forward very quickly.
- The robots can see the close beacons. The close beacons provide more information than the far beacons because they appear larger on the cplane.
- Allow the robots to see the opponents and evade them.

5.12.10: Walk phase — Obstacle avoidances

Three obstacle avoidances were developed: stuck detection (actuatorControl avoidance), stealth dog (vision avoidance) and vector avoidance (localisation avoidance).

The decision tree is depicted in figure 5.49.

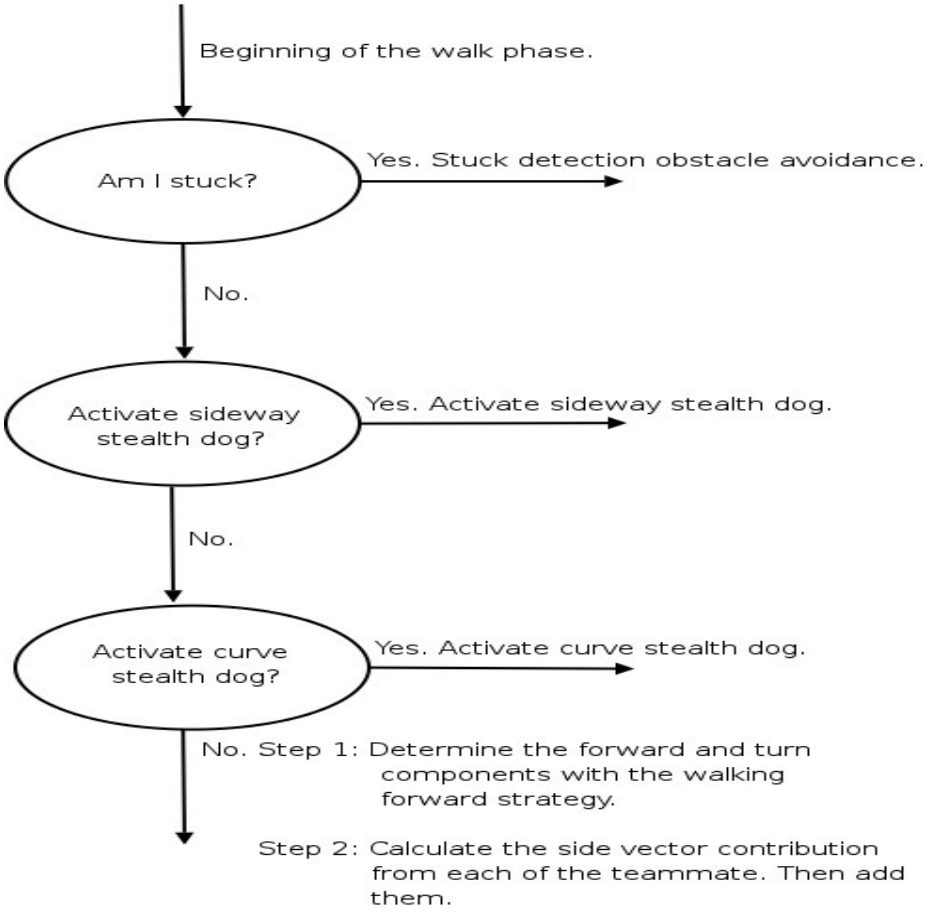


Figure 5.49: Ready player walk phase decision tree.

Stuck detection obstacle avoidance

If a ready player detects a stuck based on its PWM value, it takes few steps backward, see figure 5.50.

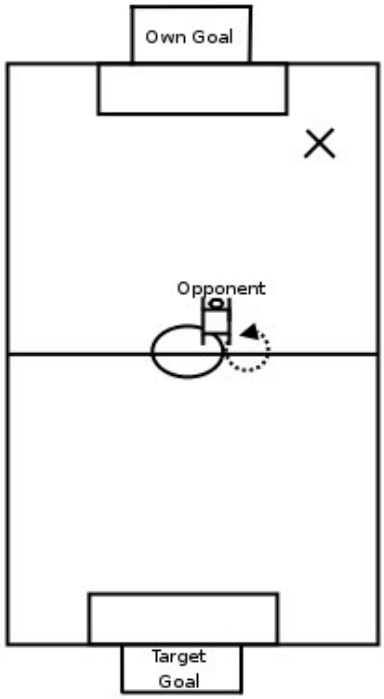


Figure 5.50: The robot detects a stuck from its PWM value. It takes few steps backward toward the direction to the kickoff position.

Borders are not recognized by the object recognition module, if the robot's localisation is misleading, the robot may not realized it is facing against the border. Stuck detection allows the robot to detect and evade the borders. Stuck detection can also detect obstructing robots in particular the opponents. For more details, refer to section 5.6.

Stealth dog obstacle avoidance

Stealth dog allows the robots to take a curved path around the opponent robots. Curve and sideway stealth dogs are both used in the ready player. Curve stealth dog is useful when the opponent robots are relatively far away. If the opponent robots are very close, sideway stealth dog is preferred. See figure 5.51. For more details including the triggers, refer to section 5.5.

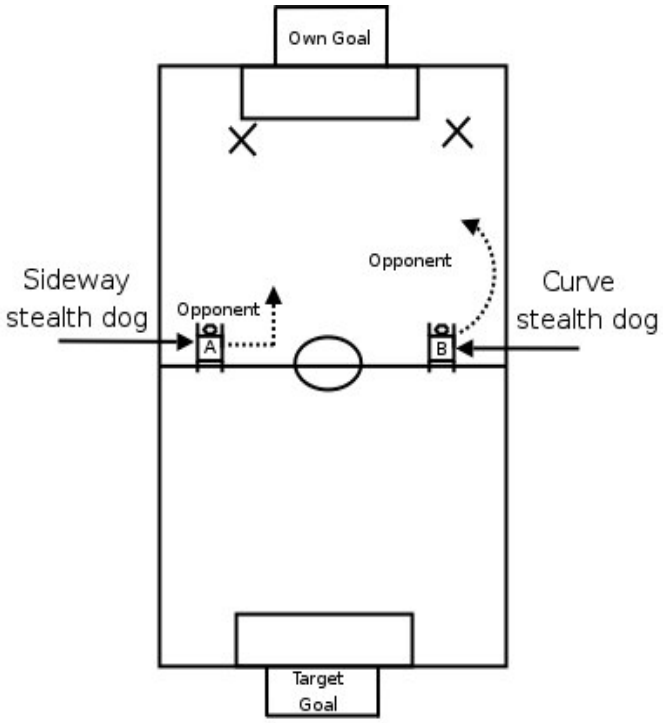


Figure 5.51: Robot A walks sideway and evade an opponent. The direction it side-steps is the direction to the kickoff position.
Robot B takes a curve locus. The direction of the curve is the direction to the kickoff position.

Vector avoidance

Vector avoidance was designed to avoid crashes between teammates. We would like the robots to stay away from each others. This is achieved by adding a small side (left/right) vector. The magnitude of the side vector is inversely proportional to the x-axis distance between the robots. This idea was suggested by rUNSWift team captain.

```
# dist is the distance between the robots.
def getTheInverseSideVector(dist):

    # alpha is a hand-calibrated value.
    alpha = 80.0

    # Return the magnitude of the side vector.
    return (alpha / dist)
```

Figure 5.52: A code fragment, side vector formula. Written in python.

The further the distance between the robots, the smaller the magnitude of the side vector. We want the robots to move away from each other quickly when they are close.

If the teammate is at the left hand side then a side vector would be added in the right direction. If the teammate is at the right hand side then a side vector would be added in the left direction. The final magnitude is the sum of each teammate's contribution excluding the goalie. See figure 5.53 and figure 5.54.

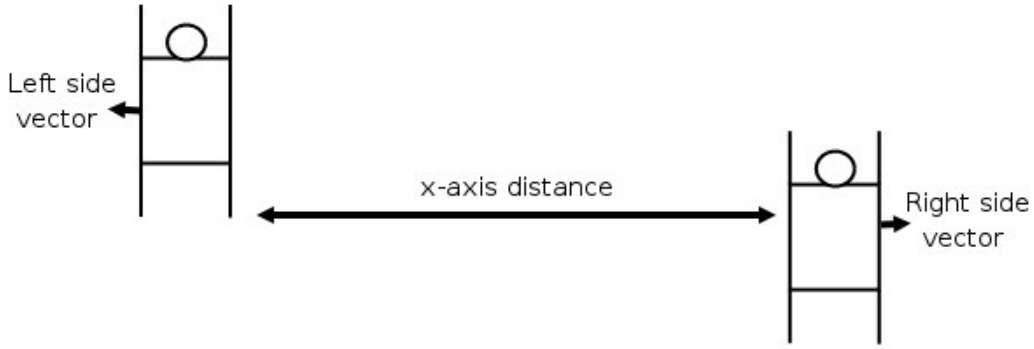


Figure 5.53: Vector avoidance. A side vector has added, its direction is the opposite of the direction to the other robot. The side vectors are small because the robots are far away.

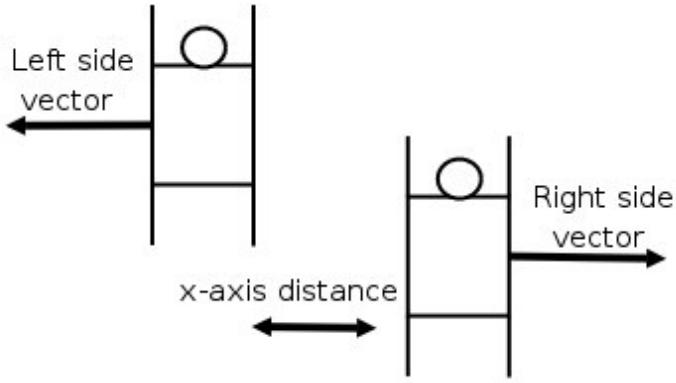


Figure 5.54: Vector avoidance. A side vector has added, its direction is the opposite of the direction to the other robot. The side vectors are large because the robots are close.

5.12.11: Phase 4 – Adjust

The robot believes it has reached its kickoff position when:

- The distance to the kickoff position is within 15cm.
- The robot's global heading is between 70 to 110 degrees (ie. Facing toward the target side).

Unfortunately robot's localisation is not perfect, sometimes the robot wants to adjust its position since its localisation has changed. Adjust is necessary when either of the following conditions satisfy, adding hysteresis:

- The distance to the kickoff position is greater than 30cm.
- The robot's global heading is not between 60 to 120 degrees.

The adjustment is usually slight. When the robots adjust themselves, they execute the phase three algorithms.

5.12.12: Putting everything together

Figure 5.55 and 5.56 present two autonomous repositioning examples.

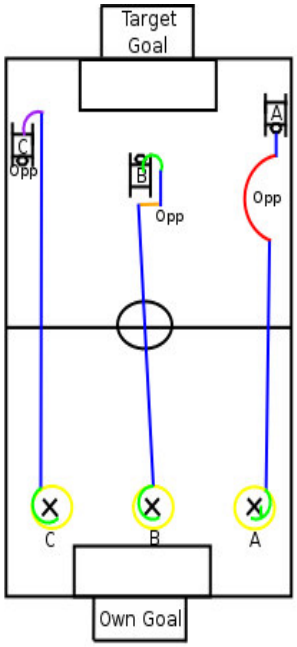


Figure 5.55: Repositioning example. Stuck detection = purple, Elliptical walk and vector avoidance = blue, normal walk = green, circle = yellow, curve stealth dog = red, sideway stealth dog = orange. Robot C detects a stuck, it walks backward and then use elliptical walk to walk straight toward position C. Robot B uses the normal walk to turn itself toward the position B. After that, it sees an opponent, since the opponent distance is small, robot B uses sideway stealth dog to walk past the opponent. Robot A sees the opponent too, but the distance is large, so it uses the curve stealth dog. When robot A, B and C reach the circle drawn around their kickoff position, they use the normal walk to turn itself toward the target side. Notice that the elliptical walk lines do not point straight directly toward the kickoff position because of vector avoidance. Robot B's elliptical walk line points straight directly toward its kickoff position because there are robot A and C's vector contribution cancel each other.

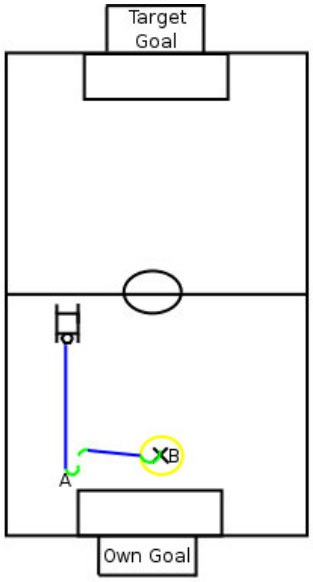


Figure 5.56: Adjust example. The robot wants to reposition at position B. Due to localisation errors, it walks to the position A. After it realises its position is not right, it walks to the position B. Blue = elliptical walk, green = normal walk, yellow = circle.

5.12.13: Performance and future development

DKPA has performed very well in this year's competition. The algorithm was able to assign kickoff positions correctly most of the time (in particular when the wireless was working). However the algorithm is not perfect — it doesn't take opponent robots into account. Opponents should be considered in the future.

The ready player was able to reposition themselves quickly in the competition. Obstacle avoidance worked reasonably well, our ready players were able to evade opponent robots most of the time. Overall the ready players performed incredibly well.

Unfortunately ready player's performance was limited by the poor vision, in particular the robot recognition. For example stealth dog requires a good robot recognition. See section 5.5 for more details.

5.13: Team Cooperation

5.13.1: Introduction

This year's team cooperation is based on the previous year, with slight modifications since it is robot independent. Currently the team cooperation codes are written in Python.

There are two forms of team cooperation in rUNSWift's strategy: global and local cooperation. rUNSwift's strategy usually assign one forward as the attacker, one forward as the supporter and one forward as the striker. The attacker attacks the ball directly. The supporter stays behind the attacker. If the attacker fails to gain possession of the ball, the supporter would switch into the attacker and attack the ball. This type of cooperation is known as local cooperation.

The furthest robot to the ball is assigned as the striker. The striker positions itself on the opposite side of the field, away from the attacker and supporter. It provides the long distance support to the attacker and supporter. This type of cooperation is

known as global cooperation. The roles are assigned dynamically based on the world model.

The idea of this strategy is aggressive. Two forwards maintain possession of the ball. The third forward stays away, waiting for opportunities to attack or defense. The following sections present an overview of the team cooperation.

5.13.2: Global cooperation

The forward furthest away from the ball than any teammate by a fixed offset is assigned as the striker. If no forward is clearly furthest away from the others, the two forwards that are furthest away from the ball are selected, the forward that is closest to the desired striker position is selected as the striker. This selection process ensures the striker reaches the striker position quickly and minimising leg-lock.

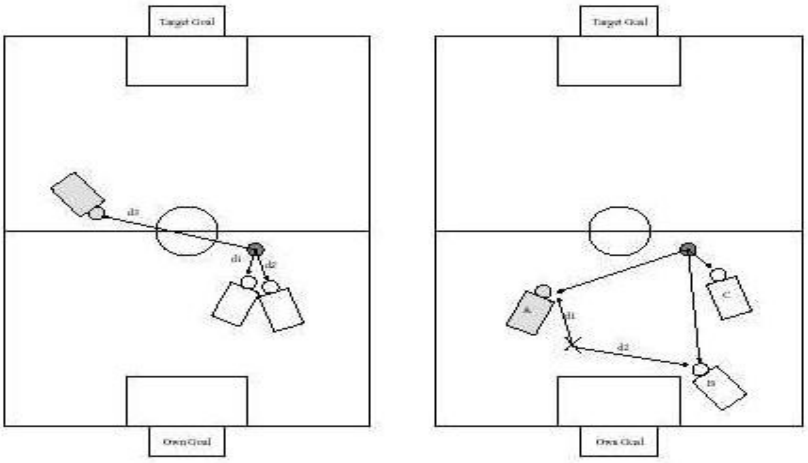


Figure 5.57: Assigning the striker. Left: Shaded robot is clearly furthest away from the ball. Right: Shaded robot is not clearly furthest away from the ball, but it is closest to the striker point. (Image courtesy of 2003 UNSW RoboCup report [13]).

Many different positioning methods have discussed in the 2003 UNSW RoboCup report (page 151 of [13]). Only the X positioning was used in the last year and this year's competition. In X positioning, four lines are drawn like in figure 5.57. If the ball is on the left side of the field, then the robot positions itself on line CE or CB, otherwise it positions itself on line CD and CA.

If the robot is behind point C (ie. one third of the field), the exact position the robot is positioned is defined by the fixed y distance behind the ball. If the robot is not behind point C, the backoff distance is related to how far the ball from point C. As the distance between the ball and point C increase, the backoff distance decreases. The idea is to position the striker closer to the ball, hence more aggressive.

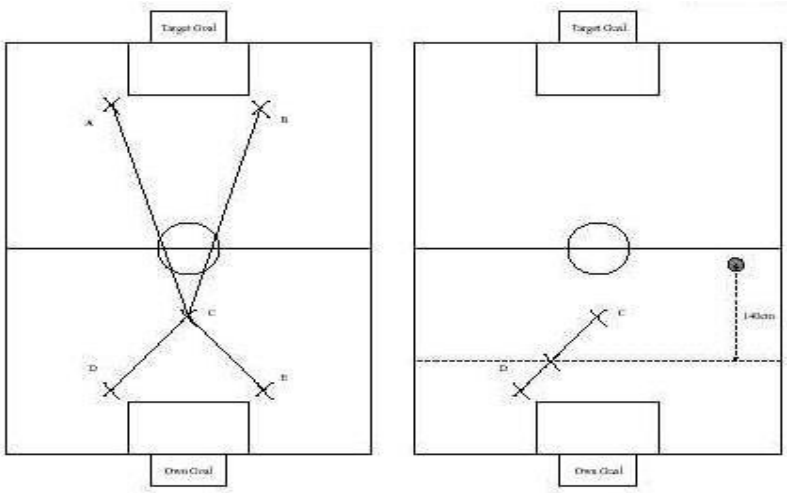


Figure 5.58: Striker's positioning. (Image courtesy of 2003 UNSW RoboCup report [13]).

5.13.3: Local cooperation

Local cooperation defines the actions taken by the two robots closest to the ball when they can see each other. A star is drawn centered on the ball, a line is drawn in the direction of the DKD (section 5.15), this line divides the star into three regions: top, middle and bottom region. See figure 5.59.

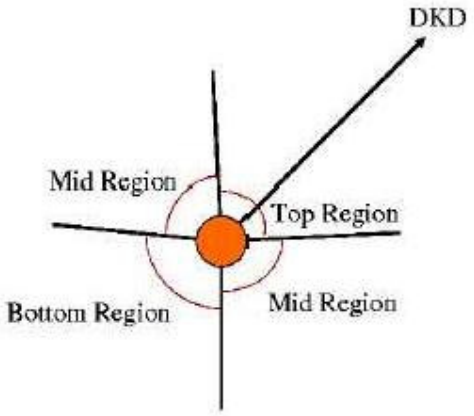


Figure 5.59: Star drawn around the ball. (Image courtesy of 2003 UNSW RoboCup report [13])

The decisions to decision which robot is in a better position to attack:

- If both robots are in the bottom region, the attacking robot is determined by the ball source, ball distance, player number and the fact which robot is closer to the DKD.

- If both robots are in the top region, the robot closest to the ball is in a better position to attack. The other robot must stay away.

- Since the bottom region is the best place to attack the ball, if a robot is in the bottom region and the other is not, the robot in the bottom region will attack the ball, the other robot must stay away.

- Otherwise, it is unsure that which robot is in a better position to attack. Both robots will use the get behind ball to break this symmetry.

The position of the supporter is depend on the ball position, robot's position and the DKD. See figure 5.60. If the ball is in top quarter of the field, the supporter will position itself at position A or B, depend on which side of the DKD line the robot is on. If the robot is on the left side of the DKD line, the robot will choose position A, otherwise position B.

Face backoff and sideway backoff are also ported into Python. Face backoff allows the robot to take a few steps backward when it detect a very close teammate. Sideway backoff allows the robot to move slowly when a teammate is beside it and chasing toward the ball.

Similarly when the ball is in the middle half of the field. However the “L” shape is wider. When the ball is in the bottom quarter of the field, the “L” shape is reversed, so that the supporter will not block the attacker.

Interested readers are referred to the last year's thesis for more details (page 151 of [13]).

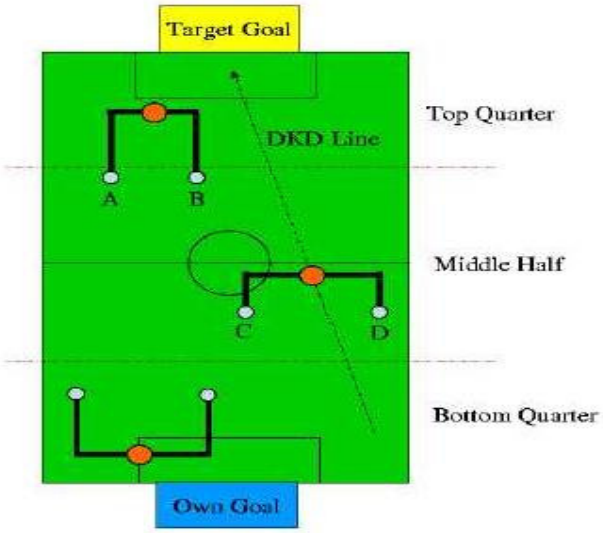


Figure 5.60: Supporter positioning.

5.13.4: Problems encountered

The team cooperation strategies remain relatively unchanged this year. Some problems encountered:

- Due to noisy camera and the fact that the two middle beacons have been removed, the robots often mislocalise in particular when a scrum has occurred. The mislocalisation completely destroy the team cooperation, since the team cooperation relies on the world model.

- rUNSWift robots have a very poor robot recognition. As a consequence they cannot backoff effectively. The robot recognition cannot recognize blue robots, hence blue robots cannot backoff at all. Although the robot recognition can recognize the red robots, but the distance, heading calculated are not accurate.

- The fact that the ERS-7 robots wear less uniform have an influence on the backoff triggers. Sometimes a robot can see its teammates for only one or few frames.

- The 2003 team cooperation decision tree is actually more complex and messy than what stated in the 2003 RoboCup report. Lots of hacking were made, the decision tree was tuned for the ERS-210 robots. It worked very well in the 2003 RoboCup competition.

However the decision tree becomes so messy that it is very hard to port it into Python.

There are some slight modifications:

- Instead of going into the supporting position when a robot is in the top region and the other robot is in the bottom region, the robot uses the get behind ball skill. Get behind ball is preferred since this skill allows the robot to turn toward the target side and stay away from the ball while backing off.

- Instead of going into the supporting position when a robot is in the top region and the other robot is also in the top region but closer to the ball, the robot uses the get behind ball skill. The reason is similar as above.

Lots of effort were put into retuning the team cooperation for the ERS-7 robots. For example changing the constants, modifying the supporting triggers. However due to limited time, the tuning is not as good as the previous year.

5.14: Wireless impact

Last year the wireless speed was slow because Sony made a bug. After the bug was fixed the wireless speed has improved from 500ms delay to approx 10 to 100ms delay. In theory, the faster wireless speed should lead to better team cooperation.

Currently the behavior in particular the team cooperation places more emphasis on the wireless than previously. For example the wireless ball and wireless teammate position are used more frequently than previously. Placing more emphasis on the wireless had some impacts on the behavior in the world open. Due to congestion the wireless speed was very slow, sometimes completely broken. Team cooperation did not performed well in the competition since it relied on the wireless.

5.15: DKD

5.15.1: Introduction

DKD stands for "desired kicking direction", it is the direction where the robot wants to move the ball. The DKD points directly toward the target side. 2003 DKD is shown in figure 5.61.

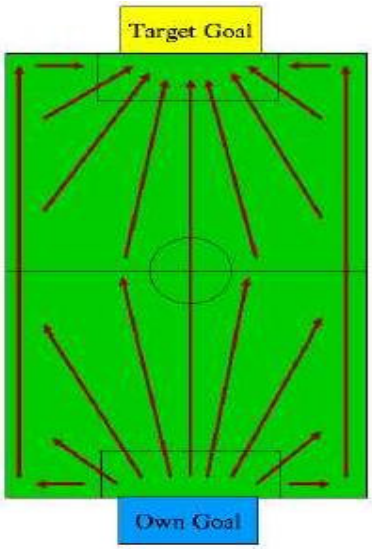


Figure 5.61: 2003 DKD. (Image courtesy of 2003 UNSW RoboCup report, page 166 of [13])

5.15.2: 2004 DKD

This year the DKD has separated into tight and wide DKD range. The new DKD is integrated with the VOAK gap finding calculation. I was responsible for integrating the new DKD with the VOAK gap finding (section 5.9), refer to [6] for details on tight and wide DKD.

If the VOAK gap finding is triggered, the DKD points directly toward the center of the largest gap. See figure 5.62. This new DKD allows the robot to kick the ball toward the gap instead of kicking toward the center of the target goal and block by the opponent goalie.

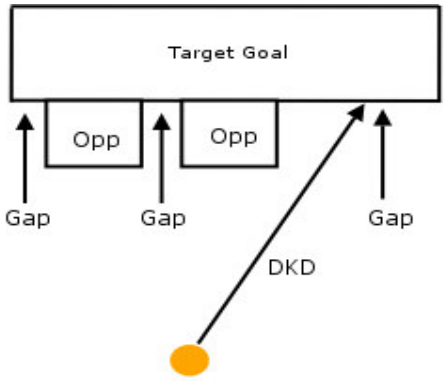


Figure 5.62: New DKD, point toward the center of the largest gap.

The following conditions trigger the VOAK gap finding calculation:

- The robot can see the ball and the target goal. Otherwise there is no point to find the gaps.
- The ball and the robot are close to the target goal. This condition ensures that the robot is in a good position to perform VOAK gap finding and kick the ball toward the gap.
- Not too many robots are blocking the way for shooting. This condition ensures that the robot has enough room to shoot through gaps.

5.16: Discussion

Problems encountered during the behavior development:

- Sometimes the position of the robots are "flipped" due to the edge detection. Refer to [7] for more details.
- Fake objects such as fake beacons, fake robots and fake goals are seen due to the inconsistency of the sanity checks. When the robots are in scrum, lots of fake objects would be seen. Refer to [5] for more details.
- The current robot recognition is poor. The blue robots almost cannot be recognized. The red robots can be recognized, but the

accuracy is not good.

- Due to poor vision, removal of two middle beacons and sometimes incorrect odometer update, the robot's localisation is not performing as well as the previous year. The robots easily get confused when they are in the corner or near the edge and can only see one beacon.

Overall the system performs much better than before the Australian Open, in particular a stable walk has been achieved. rUNSWift was able to improve the system significantly after the Australian Open.

Even with the improvements made after the Australian Open, rUNSWift was still inferior than some of the strongest teams in the 2004 RoboCup competition.

- rUNSWift's scoring ability could not match with the Germany team, UTS and Newcastle team. Our attacking speed was simply too slow, in particular when the robots were facing the edge. This is the major reason of failure against the Germany team in the round robin. During the match versus Germany, rUNSWift actually had more scoring opportunities than the Germany team. rUNSWift robots were not able to score, while the Germany team scored very quickly.

- The team cooperation obviously did not perform too well in the competition, in particular when the wireless was broken.

- Although rUNSWift robots walked much faster than in the Australian Open, some of the teams walked even faster. For example University of Pennsylvania and University of Austia robots could turned much faster than rUNSWift robots.

Bibliography

[1]. Sony Four-legged Robot league, http://www.openr.org/robocup/index.html

[2]. RoboCup Official Site, http://www.robocup.org/02.html

[3]. RoboCup Technical Committee., *Sony Four Legged Robot Football League Rule Book* (2004), http://www.tzi.de/~roefer/Rules2004/Rules2004.pdf.

[4]. RoboCup Technical Committee., *Sony Four Legged Robot Football League Rule Book* (2003), http://www.openr.org/robocup/rule/RoboCup03_rules.pdf.

[5]. Lam K.C, *Sony Four-Legged Thesis B report*, School of Computer Science and Engineering, University of New South Wales, 2004.

[6]. Chan K.C., *Sony Four-Legged Thesis B report*, School of Computer Science and Engineering, University of New South Wales, 2004.

[7]. Whaite D., *Sony Four-Legged Thesis B report*, School of Computer Science and Engineering, University of New South Wales, 2004.

[8]. Xu J., *Sony Four-Legged Thesis B report*, School of Computer Science and Engineering, University of New South Wales, 2004.

[9]. Pham K.C., *Sony Four-Legged Project report*, School of Computer Science and Engineering, University of New South Wales, 2004.

[10]. Olave A., Wang D., Wong J., Tam T., Leung B., Kim M.S., Brooks J., Chang A., Huben N.V., Sammut C., Hengst B., *The UNSW RoboCup 2002 Legged League Team*, undergraduate thesis in computer and software engineering, University of New South Wales, 2002.

[11]. Sony OPEN-R., *OPEN-R SDK Model Information for ERS-7*, Sony Corporation, 2004.

[12]. OPEN-R Official web site, http://www.open-r.org

[13]. Chen, J, Chung, E., Edwards, R., Wong, N., Hemst, B., Sammut C., Uther, W., *Rise of the AIBOs III — AIBO Revolutions* (UNSW 2003 RoboCup Team Thesis), University of New South Wales, 2003.

[14]. Set Data Structures, http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK3/NODE133.HTM

[15]. Sets, http://www.owlnet.rice.edu/~comp314/04spring/lec/week5/Sets.htm

[16]. Pham K.C, *RoboCup IT industry training report*, Taste of Research Summer Scholarship, 2004.

[17]. Kim M., Uther W., *Autimatic gait Optimisation for Quadruped Robots*, Proceedings of 2003 Australasian conference on robotics and automation, 2003.

[18]. Newcastle Robotics Laboratory, http://www.robots.newcastle.edu.au/

[19]. Best-Effort Delivery (Linktionary term), http://www.linktionary.com/b/best_effort.html

[20]. Third Australian Robot Soccer Open Championship 2004, http://magic.it.uts.edu.au/AustralianOpen2004/

[21]. AIBO SDE Homepage — FAQ, https://openr.aibo.com/openr/eng/perm/openrsdk/regi_faq4.php4#C0010

[22]. Chan K.C., Lam K.C., Whaite D., Wong T., Xu Jing., *2004 UNSW RoboCup Team Thesis A Report*, University of New South Wales, 2004.

[23]. German AIBO Team, http://www.robocup.de/aiboteamhumboldt/index.html

[24]. RoboCup German Open 2004, http://www.ais.fraunhofer.de/

GO/2004/

[25]. Sony OPEN-R message board — ers-7 crashes with dive kick, https://openr.aibo.com/cgi-bin/openr/e_regi/im_trbbs.cgi?uid=general&df=bbs.dat&prm=TAN&pg=-1&no=0998#0998

[26]. Sony OPEN-R message board — OPEN-R SDK 1.1.5-r2 Has Been Released, https://openr.aibo.com/cgi-bin/openr/e_regi/im_trbbs.cgi?uid=general&df=bbs.dat&prm=TAN&pg=-1&no=1048#1048

[27]. Lutz M., Ascher David., *Learning Python*, O'Reilly 2004

[28]. University of Pennsylvania RoboCup Team, http://www.cis.upenn.edu/robocup/index.php

[29]. Lecture 9: Kruskal's MST algorithm: Disjoint Set Union-Find, http://www.cs.ust.hk/~scheng/comp271/notes/L09.pdf

[30]. The C++ Programming Language, http://www.research.att.com/~bs/C++.html

[31]. Lutz M., Ascher D., *Learning Python*, O'Reilly, 2004.

[32]. Introduction to Pulse Width Modulation — PWM, control systems, digital control, http://www.netrino.com/Publications/Glossary/PWM.html

[33]. Sony OPEN-R., *OPEN-R SDK Model Information for ERS-220*, Sony Corporation, 2004.

[34]. Griffith University Mobile Robotics Website, http://www.cit.gu.edu.au/~s2130677/Mi-Pal/