

12-1-2003

Interactive Lego Robots Learning in a Controlled Environment

Jonathan McGehee
Western Kentucky University

Follow this and additional works at: http://digitalcommons.wku.edu/stu_hon_theses



Part of the [Other Engineering Commons](#)

Recommended Citation

McGehee, Jonathan, "Interactive Lego Robots Learning in a Controlled Environment" (2003). *Honors College Capstone Experience/Thesis Projects*. Paper 26.
http://digitalcommons.wku.edu/stu_hon_theses/26

This Thesis is brought to you for free and open access by TopSCHOLAR®. It has been accepted for inclusion in Honors College Capstone Experience/Thesis Projects by an authorized administrator of TopSCHOLAR®. For more information, please contact topscholar@wku.edu.

Interactive Lego Robots Learning in a Controlled Environment

A Thesis for the Honors Program

By

Jonathan McGehee

Directed by

Uta Ziegler

Fall 03

Approved by

Uta Ziegler
Carol W. Wilson
Markus Frey
DMG

Abstract

This paper reports on learning in autonomous Lego robots programmed to play soccer. The specific focus here is upon how different behaviors can be learned by two identical robots, based on the different types of feedback they receive from the environment. After being constructed with the same design and the same software, two Lego robots are able to perform the same basic soccer actions, such as moving to the ball, determining where the goal is, and kicking the ball. A non-deterministic version of the Q-step learning algorithm is then used to allow a robot to learn a correct series of actions based on feedback (rewards and punishment) from the environment. The goal of Q-step learning is for a robot to maximize the environmental rewards. Thus, a series of actions is “correct” if it leads to a large cumulative reward. Different environmental feedback can thus lead to a different “correct” series of actions. For instance, in this project, one of the robots learned to be an offensive player and the other robot learned to be a defensive player. This paper discusses the construction of the robots, their basic actions, the learning, training, and interactions with each other.

Acknowledgements

I would like to thank the Honors Department at Western Kentucky University for all of their contributions and assistance while working with me on my thesis project. Included with this, I would like to give special thanks to Walker Rutledge for overseeing my progress and specifically giving me feedback on this paper. I would also like to thank the Engineering department at Western for help in designing a ball for the soccer game. In the end, we did not use their design, but their contributions and efforts are greatly appreciated. Additionally, I would like to thank Terry Leeper for providing me with a facility in which to work with and train the robots. Last, but certainly not least, I would like to thank Dr. Uta Ziegler and the Computer Science Department at Western for their guidance, support, equipment, tools, and expert opinions.

Table of Contents

Introduction	1
Lego Mindstorms	2
Hardware	2
Software	6
Robot Soccer	7
Background	7
This Project	8
Rules of the Game	9
The Game	10
The Setup	10
The Robot	12
Learning	15
Basic Concepts	15
Q-step Learning	17
Results	21
Conclusions	26
Bibliography	28
Appendix A	29
The Code	29

Introduction

The purpose of “Interactive Lego Robots Learning in a Controlled Environment” is to create autonomous Lego robots that learn how to play soccer. There are many challenges to address such as how to equip the robots and their environment in such a way that the robots can interact with the environment effectively enough to play the game and still remain autonomous, which means being totally independent from any external assistance or intervention. A further challenge is to account for all real-world complexities and exceptions that might occur while playing or training. Some of these complications include such things as the gears slipping on the wheels, the battery power running down and changing the engine speed, the battery power running low on the ball and changing the range of output from the ball, and the field being uneven. Of particular interest for this project is the utilization of two different robots, each independently learning a soccer-playing strategy based on different rewards, and each interacting dynamically as “defensive” or “offensive” players.

Lego Mindstorms

Hardware

For this project the Lego Mindstorms Robotics Invention System 2.0 is used. The basis for this system, developed by a group from MIT, is the RCX brick, which is a Lego brick with a small computer in it, containing an H8300 microprocessor and 32K memory. The brick has three sensor ports for input, three motor ports to control movement, and an LCD display for output. On the front of the brick there is also an infrared port used for communication, which allows one to load programs via the Lego tower from the computer and to send information back to the computer from the brick.

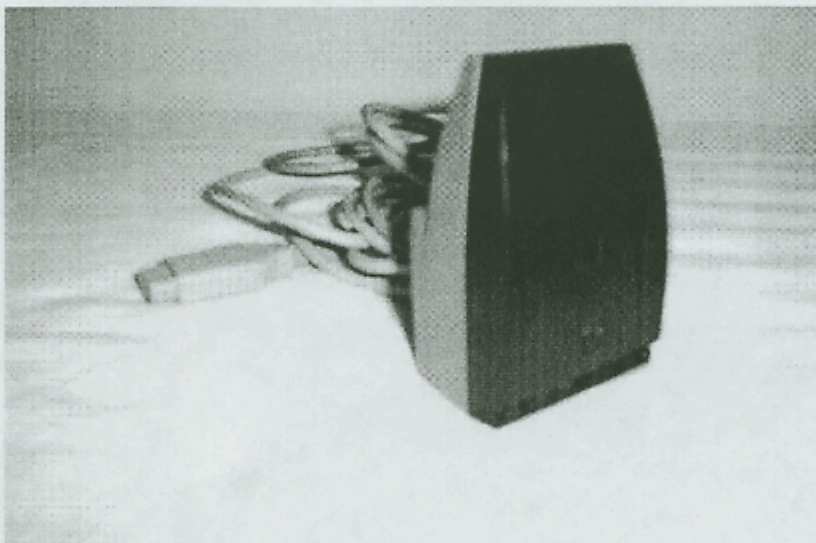


Figure 1 Lego Infrared Tower

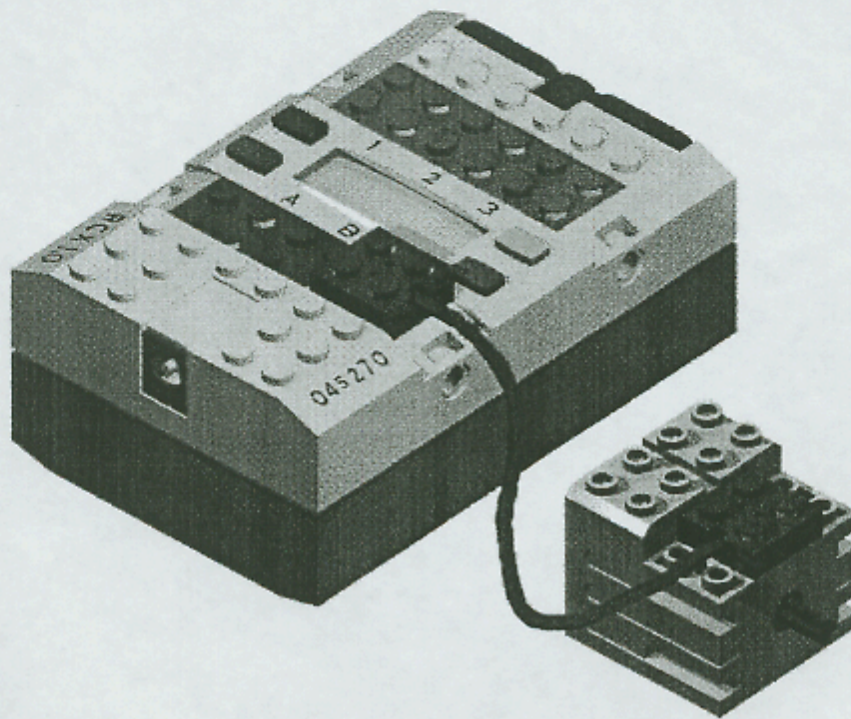


Figure 2 RCX Brick With Attached Motor

The Lego system comes with motors and a variety of sensor types—including touch sensors, rotation sensors, and light sensors. The motor is a Lego piece with a small gear coming out one side which, when connected to the brick through a wire and told to move, will spin. The motor can be used in conjunction with other pieces such as gears, wheels, and tracks to move the robot or some part of the robot. The motor can go forward, backward, brake, and have its speed set to a value between 0 and 255, with 255 being the fastest.

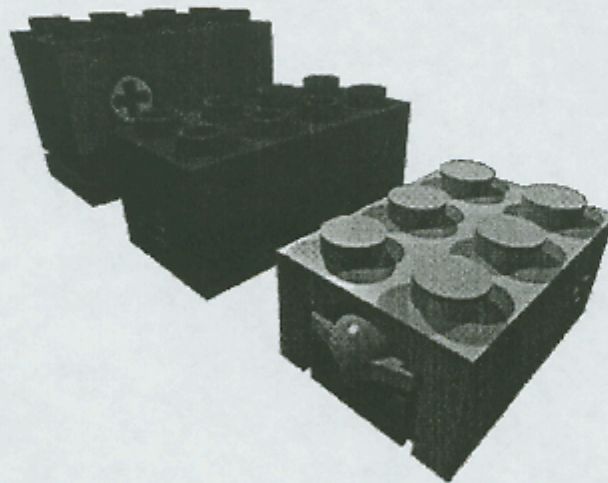


Figure 3 Rotation, Light, and Touch Sensors

The touch sensor is a small Lego piece with a depressible yellow button on the front.

With this sensor attached to one of the input ports on the brick, the robot can detect when the button is depressed because the sensor will give an input of either 0 for “not depressed” or 1 for “depressed.” It is commonly used to detect obstacles such as walls or other robots.

The rotation sensor is another Lego piece with a hole mount in it so that a rotating piece can be put through it to detect how many rotations the device has made. This information is relayed to the brick in the form of clicks. An entire rotation of 360 degrees is registered as 16 clicks. This sensor is used mostly to detect wheel movements for distance or to keep the movements parallel.

There is the light sensor. It is another Lego piece with a sensor on the front as well as its own LED. The light sensor returns a value somewhere between 0 and 100 according to the intensity of light detected, with 100 being the brightest. The light sensor can be used to sense any kind of light source or to measure light intensity. There is also an infrared light sensor, which detects infrared light much the same way as the light sensor detects visible light. This sensor did not come with the Lego Mindstorms set, but was ordered separately.

In addition to the brick, motor, and sensors, the set includes the regular Lego pieces for construction.

Software

In order to write programs and utilize the capabilities of the Lego Mindstorms, one needs an environment in which to write the code, compile it into an executable file, and send that information over the COM port infrared tower to the robot. In this project, BrickOS is used as the operating system for the Lego brick. This provides the programmer with an interface to control the motors and sensors, the possible use of multiple threads through scheduling and interrupts, as well as basic control of the program execution. Since BrickOS only runs in a UNIX environment, while this project utilizes Windows-based machines, Cygwin, which is a UNIX emulator, is downloaded and installed onto the computer. Then the necessary compiler is installed so that the C/C++ code can be compiled into H8300 code. Next, the BrickOS is installed under Cygwin, which contains a) code for the kernel which will be compiled and downloaded to the brick for its operating system, and b) code which allows a C/C++ program to interface with the sensors, motors, or other pieces of equipment attached to the brick by giving it methods that provide access to the full functionality of the various devices. Other alternative environments that could be used include the LEGO operating system that comes with the Mindstorms set, Not Quite C, and leJOS, which supports Java. BrickOS was selected for this project because it was a familiar package and because it offered support for C/C++, which was also well-known.

Robot Soccer

Background

The initial concept for this project was taken from the Robocup competition, which involves groups working with soccer-playing robots. The reasons for choosing soccer are because it is a team game and because it is dynamic in nature. These are topics of interest because a team game allows for different robots to interact and work together, and the fact that the game is dynamic, or constantly changing, forces the robots to be able to handle real-world variability in real-time. There are several different characteristics that these other projects have. For instance, the Robocup competition uses an overhead camera that takes pictures of the field, showing the location of the robots, the ball, and the goals. Then the image is processed, and the information is forwarded to the robots, providing them all with the knowledge about where things are located (<http://www-2.cs.cmu.edu/%7Ebrettb/robocup/index.html>). Others have also used a wide variety of alternative constructions such as the Sony AIBO robot, which is a robotic dog (Trivedi), or the BSR-Soccer mobile base, which is omni-directional mobile base developed for the Robocup competition (<http://www.mega-robot.com/>).

This Project

The present soccer game takes on this challenge but has several characteristics that make it different. In contrast to the Robocup robots, the Western Kentucky University robot keeps track of locations internally and uses its own sensors, making it truly autonomous. Also the Western Kentucky University robot kicks the ball, requiring that it has to find the ball again instead of holding onto the ball. This creates a more realistic game in which an opponent might have an opportunity to take away possession. Yet another difference is in the construction of the robot. The WKU robot is constructed using the LEGOS Mindstorms Robotics Invention System 2.0 as discussed earlier. Finally, the main point of emphasis for the WKU robot that makes it different from others is that it learns how to play on its own. Here, the robot learns to coordinate actions to achieve its goal instead of being hard-coded to know or told explicitly. In theory, a robot implemented in this fashion wouldn't have to follow an exact sequence of actions, which might work in some instances but fail in others. A robot based on this learning algorithm can adapt while in play, giving it a significant edge over other non-learning robots.

Rules of the Game

The rules of this game are few. The offensive robot is to try to score a goal while the defensive robot is just trying to keep the offensive robot from scoring. In any extenuating circumstance in which any robot or the ball gets stuck on a wall, corner, or other robot, the operator will intervene and reset the game.

The Game

The Setup

There are two parts to the setup of this game. The first is the “soccer field” or playing field, which is six feet by four feet with a goal at each end and a light above each goal. The field itself is made out of wood and is constructed so that it can fold up for transport. The field also has walls around it so that neither the robot nor the ball is able to get off the field. The floor of the playing field is divided up into four regions: each half of the field and then a small area in front of each goal. Each one of these areas is painted a different color.

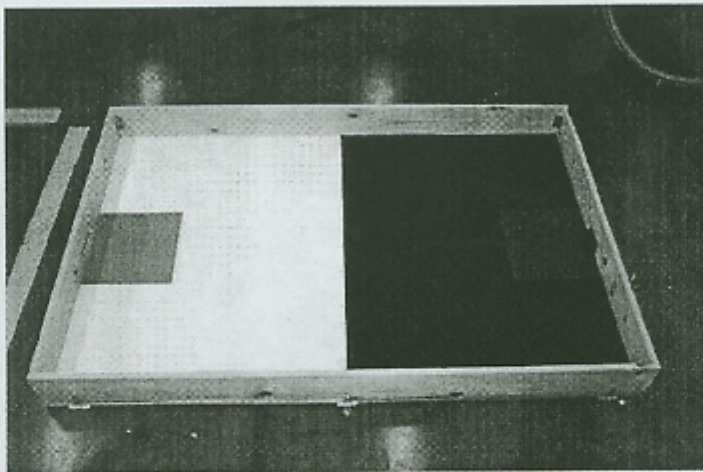


Figure 4 Soccer Field

The ball used in this project was manufactured by Wiltronics Research Pty. Ltd. It is made of a clear plastic casing with a circuit board powered by a rechargeable nickel-cadmium battery inside. Extending out from the circuit board on all sides are small infrared emitting diodes.

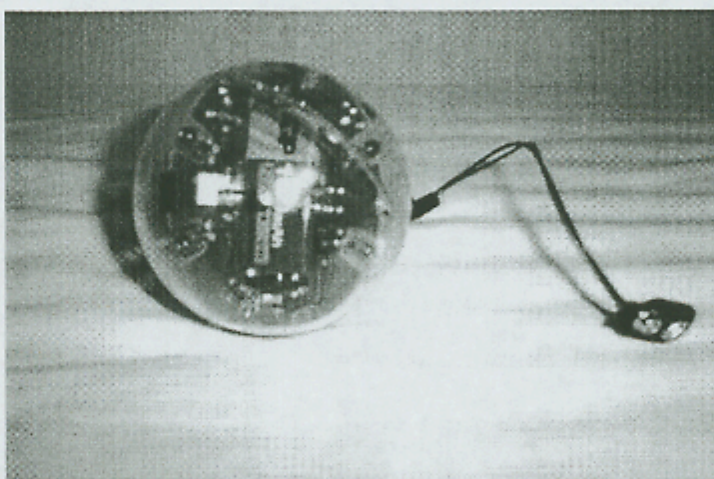
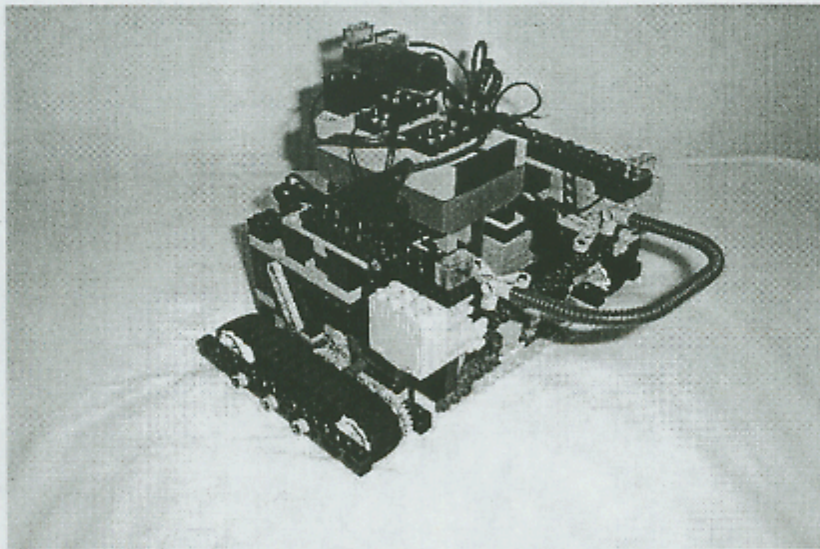


Figure 5 Roboball



The Robot

With all these elements prepared, the soccer robot can be constructed. Since the structure of the robot determines its function, it is essential that all the tasks that have to be accomplished by the robot be addressed.

First, the robot needs to move in a predictable fashion. Otherwise, many of the underlying assumptions that the robot has to make could be false. After several designs and models were constructed and rejected, one was built which uses tracks for movement since they seem to be the only way to get the robot to move in an expected manner.

Another task necessary for the robot is the ability to gain possession of the ball and subsequently kick it. In order to create this functionality, the tracks are set out wide, creating room for a kicking/grabbing mechanism, which is set in between them. Motors run both the tracks and kicking/grabbing mechanism. The top segment of the kicking/grabbing mechanism is a grasping device that allows the robot to hold the ball as it spins so that the ball is not lost. Also, the bottom segment of the arm is used as a kicking device.

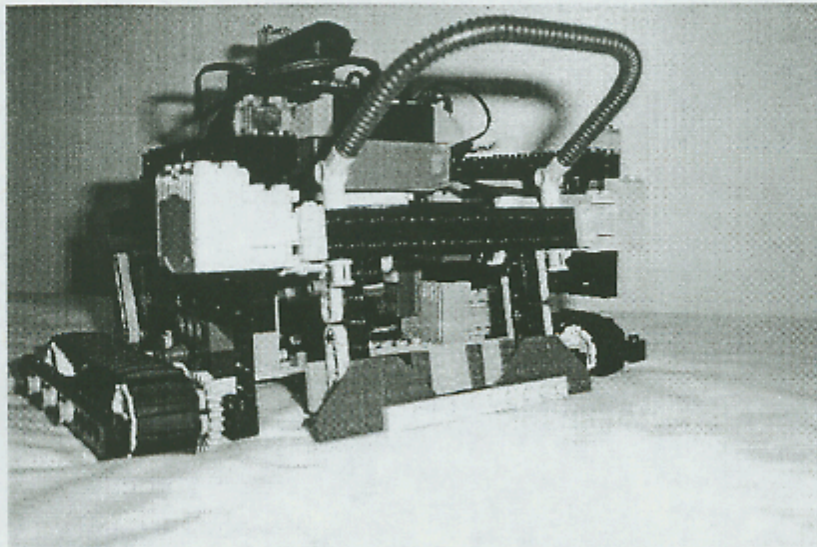


Figure 6 Front View of Robot With Kicking and Grabbing Mechanism

Another task of the robot is to determine the location of the goal. This functionality is provided by a light sensor, which is mounted on top of the robot facing forward. It is also wrapped with construction paper in order to give this sensor “tunnel vision.” This is necessary so that the robot will align precisely with the goal.

Also mounted on the top of the robot is a touch sensor that allows the robot’s “trainer” to give the robot rewards. This is done simply by depressing the touch sensor button when prompted and after the robot has performed a desirable action.

Yet another function required by the robot is the ability to determine its location on the field. There is an additional light sensor mounted beneath the robot that faces the ground and enables the robot to detect this by means of the different colors painted on the field’s surface.

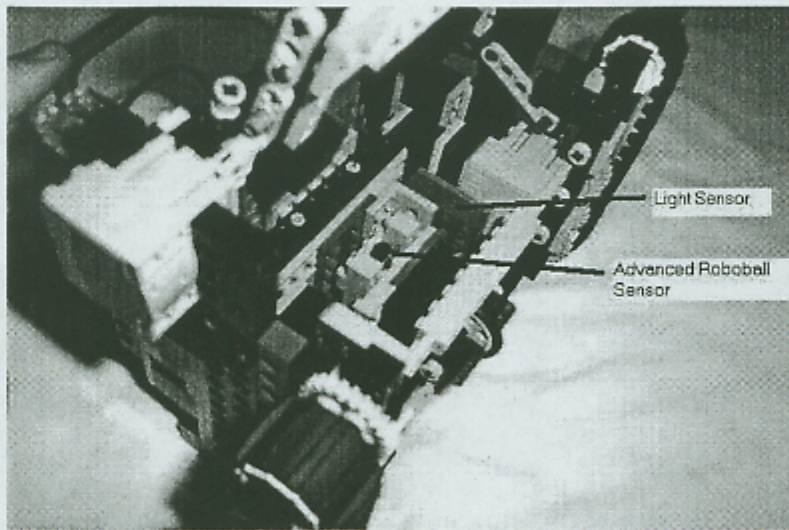


Figure 7 Underneath View of Robot

Finally, the robot must find and go to the ball in order to gain possession and kick it. For this there is the Advanced Roboball sensor from Wiltronics Research Pty., which enables the robot to detect the infrared light emitted from the ball.



Figure 8 Advanced Roboball Sensor

Learning

Basic Concepts

For the learning algorithm used here, the robot must have a previously defined set of actions from which to select. It selects the actions according to which one will help it reach its goal, which is based on the rewards it receives. The set of actions for this robot are a “defensive find the goal,” an “offensive find the goal,” a “dribble,” a “shoot,” and a “find the ball.”

The “defensive find the goal” is made so that the robot will turn in the direction of the goal. In this instance it does not attempt to align itself precisely because a defensive player will sacrifice accuracy for a quicker response. The only important thing to the defensive player is that he is directing himself away from his goal so that he can kick the ball away from his goal.

The “offensive find the goal” works exactly like the “defensive find goal” at first in that it turns in the general direction of the goal. However, it is different in that it will then attempt to align itself precisely with the goal it wants to shoot to. Obviously this is necessary for accurate goal shooting.

“Dribble” is just like real dribbling in soccer. The robot kicks lightly with its kicking arm in order to move the ball down the field and get into position to shoot at the goal.

“Shoot” works the same as “dribble” by having the robot kick with the kicking mechanism, but the robot does it more aggressively with a harder kick. This is designed for strong, quick goal shooting.

Finally, there is “find the ball,” which works much like the “offensive find the goal” by instructing the robot to turn in the general direction of the ball and then attempt to precisely line up with the ball. The robot, however, goes to the ball and takes hold of it with the grabbing mechanism.

The robot also has a set of predefined states that are delineated by a) field location and b) goal direction from the front of the robot.

Q-step Learning

The Q-step method of learning is a form of reinforcement learning, meaning that as the robot plays, it is either given rewards for its actions, reinforcing its selection, or it is not given rewards, thus not reinforcing the selection.

The basic algorithm for Q-step Learning is as follows:

- 1) Select an action
- 2) Perform the action
- 3) Observe the new state
- 4) Update the Q_scores matrix
- 5) Update the state
- 6) Loop

The first step of selecting an action is somewhat more complex than it seems. Although at the beginning of training the selection of an action is quite random, as the robot learns, it uses what it learns to choose the best sequence of actions. However, at what point during the learning does the robot begin to assume there is no more experimenting or random selection necessary that could lead to other alternatives that might be better than what it has learned? This is the problem of experimentation versus exploitation, which is solved by using the following conditional probability:

$$P(a_i | s) = \frac{k^{Q_scores(s, a_i)}}{\sum_j k^{Q_scores(s, a_j)}}$$

$P(a_i | s)$ is the probability that one will choose action a_i if he is in the state s . We calculate this value for each action from that state by taking the given constant k to the power of what is in the Q_scores matrix for that action and state divided by the sum of k to the power of what is in the Q_scores matrix for each action from that state. For example, if a given row of the Q_scores matrix has the values 2, 3, 5, and 1 for each of the four possible actions and the value of k equals 2, then the probability that the first action is selected would be

$$P(a_1 | s) = 2^2 / 2^2 + 2^3 + 2^5 + 2^1 = 4/46$$

Then one takes each of these values for each action and gives that action a percentage of the values between 1 and 0. Finally, a random value p is selected such that $1 \geq p \geq 0$. Then the action with the greatest percentage of values will be the most likely to be selected. Also, the given value of k is essential to this problem of experimentation versus exploitation. If in this above example one were to make the value of k equal to 5, and do the calculations over, he would see that the largest portion designated to the action with the value of 5 would grow from 69.6 percent to 95.3 percent. This would obviously make it much more likely that that action would be randomly selected. Consequently, the larger the value of k , the more one leans toward exploitation. The smaller the value of k , the more he leans toward experimentation (Mitchell).

After choosing the action, the robot performs the action and observes its new state. In the present case, the state of the robot is determined by a couple of factors, the first of which is where the goal is in relation to the front of the robot. It can be to the front, the back, or the front-left, for example. The other factor is which one of the four areas the robot is in. It can be in its offensive end, defensive end, offensive goal, or defensive goal. By using every possible combination of these two factors a total of thirty-two states can be defined.

With these states and the previously defined actions, the agent generates a Q_scores matrix with the rows representing the states and the columns representing the actions. Initially the matrix is set to all zeros. The purpose of the matrix is to keep track of the estimated cumulative reward for each action from each state. For example, the value in row one and column one is the estimated cumulative reward for being in state one and doing action one. The basic equation used in updating the reward matrix is as follows:

$$Q_scores(s,a) = r(s,a) + \gamma * \max_k Q_scores(s',k)$$

s is the state before the action, a is the action performed, s' is the new state after the action, Q_scores is the estimated cumulative rewards matrix, and $r(s,a)$ is the immediate reward for taking action a from state s . Then $\max_k Q_scores(s',k)$ is the maximum reward that can be achieved from the new state s' , which is whatever action k that is in the row s' with the highest estimated cumulative reward. Since this reward is not immediate, it is scaled by γ , which is a constant such that $1 \geq \gamma \geq 0$. Finally, after these updating calculations are performed, the agent resets its current state to the new state and loops again to select the next action (Mitchell).

This calculation is a somewhat simplified version of what this project has to consider.

The above equation for finding the cumulative reward works under the assumption that the reward and next-state functions are deterministic. In other words, it is true every time that if the robot is in state one and does action one, it ends up in state four, for example. This would make certain what the next state would be, and then one could also set the rewards precisely. However, this project is working under non-deterministic conditions, so some modifications have to be made. The equation for this is as follows:

$$Q_scores_n(s,a) = (1-\alpha_n)Q_scores_{n-1}(s,a) + \alpha_n [r(s,a) + \gamma * \max_k Q_scores(s',k)]$$

where:

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

$Q_scores_n(s,a)$ is the value being updated. $Q_scores_{n-1}(s,a)$ is the value at that position previously. $\text{visits}_n(s,a)$ is the number of times that the position (s,a) has been visited up to and including the current time. One scales the previous value in the matrix at that position by $1-\alpha_n$ and the current rewards by α_n . What is done here is that one changes the cumulative reward values more slowly, taking into consideration earlier values. For example, if the position is only visited once, then the value of α_n will be one-half; therefore, the weight of the previous value will be one-half, and the weight of the current value will be one-half for the weighted average. As one visits the position more often, the value of α_n will decrease, and more weight will be given to the previous value and less to the current value, a procedure which offers protection from giving too much weight to some bizarre occurrence. Also, it should be noted that if the value of α_n is 1, then the value of the above equation is equivalent to the deterministic equation (Mitchell).

Results

After training both the offensive and defensive robots for approximately 500 iterations, the resulting Q_scores matrices can be extracted and examined. The offensive robot's matrix is as follows:

FB	OF	DF	D	S
0	0	0	0	0
37.5	0	0	49.22	25
83.21	0	37.5	0	37.5
96.88	0	0	0	0
0	0	25	0	25
25	0	0	0	0
94.53	0	0	93.75	93.5
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
90.14	0	46.88	0	43.75
0	0	0	0	0
0	0	0	0	0

0 0 0 0 37.5

0 0 43.75 0 93.5

0 0 0 0 25

0 0 0 0 0

0 0 0 0 0

99.17 0 0 93.75 25

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

42.11 93.75 0 0 93.5

0 0 0 0 43.75

0 0 0 0 0

0 0 0 93.75 0

0 91.95 0 93.75 43.75

0 0 0 0 0

0 0 0 0 0

0 0 0 0 25

83.24 0 0 93.75 43.75

0 0 0 0 0

Each row in the matrix represents one of the thirty-two possible states. Each column of the matrix represents one of the actions. FB stands for find ball, OF for offensive find goal, DF for defensive find goal, D for dribble, and S for shoot. By looking at the high numbers in the matrix, one can determine the most highly regarded actions from particular states. These actions from these states should be the most likely to be repeated and are the most rewarded ones. In the case of the offensive robot, these include find ball for states four, seven, and nineteen. The matrix also shows a high affinity for dribble for states four, nineteen, twenty-six, twenty-seven, and thirty-one. The exact definition for each state can then be determined as well, resulting in a thorough understanding of what the robot has learned. For example, state four coincides with being in the offensive goal region on the field with the goal in front of the robot. Therefore, the robot learned that from this state, it would be desirable for the robot either to dribble or to find the ball. Additionally, one can determine from which states the robot learned to perform a specific action. For instance, if a researcher wanted to know from what state the robot learned shooting was a good choice, he could look and see that state seven had a high reward for this action. This state means the robot was in the defensive goal region with its goal toward the front-right. This would seem to be a good action because it would forcefully remove the ball from this potential point-losing position and send the ball toward its goal-scoring end.

Here is the learned reward matrix for the defensive robot:

FB	OF	DF	D	S
0	0	0	0	0
60.16	0	37.5	0	50
67.36	0	43.75	0	87.5
0	0	0	0	0

32.81	0	0	48.44	93.5
25	0	43.75	0	50
68.08	0	25	0	93.5
0	0	0	0	0
31.25	0	0	46.88	98
0	0	43.75	0	0
86.90	0	0	29.17	93.5
0	0	0	0	0
12.5	0	0	0	0
46.88	0	43.75	0	93.5
86.98	0	43.75	0	93.5
0	0	0	0	0
25	0	0	0	0
45.63	0	43.75	0	93.5
90.38	0	43.75	0	93.5
0	0	0	0	0
0	0	0	0	50
0	0	0	29.17	0
62.24	0	43.75	0	93.5
0	0	0	0	0
0	0	0	0	0
46.88	0	0	0	93.5
81.73	0	43.75	0	93.5

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
59.18	0	0	93.75	93.5
0	0	0	0	0

After analyzing this data in the same manner, one can see that there are high levels of rewards for shooting from state five, seven, nine, eleven, thirteen, and several others. In fact, it appears that shoot contains many high values. Other notable values include “dribble” for state thirty-one--in the defensive goal region with the robot’s goal to the front-left--and “find ball” for state nineteen--in the defensive goal region and facing the defensive goal.

Conclusions

From the results, one can see that the robots did learn some correct actions to take in various circumstances. Although the robots did not perform the best and most efficient series of actions every time, they did at crucial times learn to perform well. From an observational standpoint, the robots noticeably repeated actions they were rewarded for, and thus there was a marked difference between each separately trained robot because of their differing rewards. The offensive robot performed quite well and scored several goals during training. The defensive robot did not seem to perform as well. It is, however, necessary to recognize that the criteria for a "successful" defensive robot are more vague and subjective than the obvious sign of success for the offensive robot.

The final interaction between the offensive and defensive robots--after they had been trained and then put in a game against each other--produced some notable results. One point of interest is that the robots interacted and competed for possession of the ball without any form of avoidance procedure. In other words, the robots were not equipped with any sort of method to deal with getting stuck against the wall or another robot. The reasons behind this stem from both the nature of the program and the structure of the robots. The program contributes to this fact because the only time the robots will move forward is when they are going after the ball. Therefore, if the robots are going after the ball and run into some obstacle, the ball will move to

the side out of range of the infrared sensor, and the robots will stop going forward and then scan the area for the ball, re-orientating themselves into an open area. Another function of the program that contributes is the fact that the robots spin on the spot versus turning. This action allows the robots to deflect or slide past some obstacle instead of becoming caught. The structure of the robot works well in avoiding a trapped situation because of the compact design with few projecting extremities. The only component of the robot that protrudes is the grabbing mechanism, which is very flexible. This allows the arm to slide off or around any impediment.

Another observation that was made during the two-robot interaction was that one robot could gain possession and shield the other robot from being able to see the ball. This reduced the interactivity at some points because one robot would have the ball and be acting on it while the other robot was spinning in an effort to determine the ball's location. But this was not the case all of the time. There were several times where the robots were fighting over control of the ball, and possession would shift from one robot to the next. There were even a few times that both robots grabbed the ball simultaneously.

As with most learning algorithms, these robots could have benefited from additional training. The more iterations these robots could run with proper observation and rewarding, the more efficient their play would become.

In summary, this design, set up, and learning process seem to be very effective for developing a set of interactive robots with their own strategies for playing soccer.

Bibliography

Artificialia. "RoboCup Junior." Online posting. 18 Oct. 2001.

[<http://www.artificialia.com/RoboCupJr/>](http://www.artificialia.com/RoboCupJr/).

Lego Mindstorms. Online posting. [<http://neuron-ai.tuke.sk/~jeppe/msrobots.html>](http://neuron-ai.tuke.sk/~jeppe/msrobots.html).

MegaRobot: Robocup Robot Hardward. [<http://www.mega-robot.com/>](http://www.mega-robot.com/).

Mitchell, Tom M. Machine Learning. Boston: WCB McGraw-Hill, 1997.

Nilsson, Nils J. Artificial Intelligence: A New Synthesis. San Francisco: Morgan Kaufmann Publishers, Inc., 1998.

Smallsize Robocup 2002, Fukuoka Japan June 19-25 [<http://www-2.cs.cmu.edu/%7Ebrettb/robocup/index.html>](http://www-2.cs.cmu.edu/%7Ebrettb/robocup/index.html).

Trivedi, Bijal P. 'Soccer Robots Compete for 6th Annual Robocup.' National Geographic Today, June 17, 2002. Web. News.nationalgeographic.com.

Appendix A

The Code

```
#include <conio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dsensor.h>
#include <dsound.h>
#include <dmotor.h>
#include <dlcd.h>
#include <time.h>
#include <persistent.h>
#include <lnp/lnp-reliable.h>
/*****
//CONSTANTS(in all caps)
/*****
//Here are defined most basic constants of the program
#define NUM_OF_GOAL_LOCS 8
#define NUM_OF_ROBOT_LOCS 4
#define NUM_STATES 32
#define NUM_ACTIONS 5

//Here we define somemore constants
const int MOTORSPEED = 150;
const int DRIBBLESPEED = 25;
const int SHOOTSPEED = 75;
const unsigned int SEE_BALL_THRESHOLD = 15;
const unsigned int FOUND_BALL_THRESHOLD = 95;
const unsigned int SEE_GOAL_THRESHOLD = 50;
const float Y = .8;
/*****
*****

//TYPEDEFS and ENUMERATION TYPES
/*****
*****

//These two typedefs define these variable types with some more readable names
```



```

typedef int turn;
const turn RIGHT = 0;
const turn LEFT = 1;

typedef int bool;
const bool true = 0;
const bool false = 1;

//These enumerations define three important pieces of information in a more readable fashion
enum b_loc {ftr, ftrright, right, bckright, bck, bckleft, lft, ftrleft} ball_location;
enum g_loc {front, frontright, right, backright, back, backleft, left, frontleft} goal_location;
enum field_loc {defensive_end, offensive_end, defensive_goal, offensive_goal} robot_location;
//*****
//GLOBAL VARIABLES
//*****
//These are the global variables. "spin_threshold" is used in both the spin function and the
wait_event function
//for seeing the light. Since it must maintain its value from one function to the other, it must be
global. "time"
//is used in nearly every function and is very naturally global. "k" is a number that varies
throughout the program
//that determines whether the robot experiments more or exploits what it has already learned.
"reward" is used in
//many of the functions concerning the internal calculations. The entire learning process is based
greatly on this
// value. Finally we have the values "current_state", "new_state", and "action_selected." All of
these values are
//updated and used in the calculation functions every time through the learning loop.
unsigned int spin_threshold;
int time;
int k = 2;
float reward;
int current_state;
int new_state;
int action_selected;

//*****
//These arrays, which are global as well, contain very important data in relation to the learning
process. Q_scores
//contains values the program uses to determine the next action. Also the visits array is used to
determine how far
//along in training the robot is. Both of these are made persistant so that in between training
periods, this critical
//data is not lost.
const int state_array[NUM_OF_GOAL_LOCS][NUM_OF_ROBOT_LOCS] =
{{1,2,3,4},

```



```

{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},
{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},
{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},
{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},
{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0}};

//*****
*****

//for debugging
int action_array[30];
static const note_t song[] = {{PITCH_G4, QUARTER},{PITCH_F4, HALF},{PITCH_G4,
QUARTER}}};

//*****
//THREADS
//*****
//Here we define the only thread besides the main thread. It is used to get the reward
//from the trainer.
pid_t reward_thread;

//*****
//MOTOR MOVEMENTS
//*****

//function runs the motors
//only called by other motor control subprograms
void run_motors(){
    motor_a_speed(MOTORSPEED);
    motor_c_speed(MOTORSPEED);
}

//set the direction for the motors to forward and run for "time" milliseconds
void go_forward(int time){
    motor_a_dir(fwd);
    motor_c_dir(fwd);
    run_motors();
    msleep(time);
}

//set the direction of the motors such that the robot will spin to the
//left or counterclockwise for "time" milliseconds
void spin_left(int time){
    motor_a_dir(rev);
    motor_c_dir(fwd);
    run_motors();
    msleep(time);
}

```



```

}

//set the direction of the motors such tha the robot will spin to the
//right or clockwise for "time" milliseconds
void spin_right(int time){
    motor_a_dir(fwd);
    motor_c_dir(rev);
    run_motors();
    msleep(time);
}

//stops the motors for "time" milliseconds
void stop_motors(int time){
    motor_a_speed(0);
    motor_c_speed(0);
    motor_a_dir(brake);
    motor_c_dir(brake);
    msleep(time);
}

//uses the arm to hold the ball
void grab_ball(){
    motor_b_dir(rev);
    motor_b_speed(25);
}

//release the ball the arms grip
void release_grip(){
    motor_b_dir(brake);
    motor_b_speed(0);
}
//*****
//WAIT_EVENT FUNCTIONS
//*****

//this function is used in conjunction with the 360spin function
//to stop the spin when it sees the light with the LIGHT_1 sensor
wakeup_t see_light(wakeup_t dummy){
    return (LIGHT_3 > (spin_threshold + 20));
}

//This function is used for two purposes. First it is used to determine whether we are using the
robot
//in a training mode or in a playing mode. If the touch sensor is touched, then the robot will be in
//playing mode and will not wait for the trainers reward inputs. Secondly, this function is used as
the

```



```

//indicator of whether or not the robot got a reward for the last action it performed. If the touch
//sensor is touched, then the robot got a reward, otherwise there was no reward.
//Technical: the reason for the 120 in the boolean statement is we had a touch sensor stacked on a
//light sensor. This value then allowed us to know that since the max light reading is
approximately
//100-102, the touch sensor had to have been activated to produce such a high number.
wakeup_t touched(wakeup_t dummy){
    return(LIGHT_1 > 120);
}

//*****
//MATHEMATICAL OPERATIONS
//*****
//returns a random number between 0 and 1
float gen_random(){
    float temp;
    temp = random();
    temp /= 2147483647;
    return temp;
}
//computes a number to a power
float power(int base, float exponent){
    int temp = base;
    int z;
    for(z=1; z<exponent; z++)
        temp *= base;
    return temp;
}

//*****
//DYNAMIC UPDATING OF IMPORTANT VARIABLES
//*****
//This function is called everytime the robot moves in order to update and maintain an accurate
//memory of where the ball is approximately located. That way if the robot then needs to find
//the ball, he should be able to get properly oriented.
void update_ball_position(turn direction, int change){
    if(direction == RIGHT){
        if(change == 45){
            if(ball_location == ftr)
                ball_location = ftrleft;
            else if(ball_location == ftrright)
                ball_location = ftr;
            else if(ball_location == rgt)
                ball_location = ftrright;
            else if(ball_location == bckright)
                ball_location = rgt;
        }
    }
}

```



```

        else if(ball_location == bck)
            ball_location = bckright;
        else if(ball_location == bckleft)
            ball_location = bck;
        else if(ball_location == lft)
            ball_location = bckleft;
        else if(ball_location == frtleft)
            ball_location = lft;
    }
    else if(change == 90){
        if(ball_location == frt)
            ball_location = lft;
        else if(ball_location == frtright)
            ball_location = frtleft;
        else if(ball_location == right)
            ball_location = frt;
        else if(ball_location == bckright)
            ball_location = frtright;
        else if(ball_location == bck)
            ball_location = right;
        else if(ball_location == bckleft)
            ball_location = bckright;
        else if(ball_location == lft)
            ball_location = bck;
        else if(ball_location == frtleft)
            ball_location = bckleft;
    }
    else if(change == 135){
        if(ball_location == frt)
            ball_location = bckleft;
        else if(ball_location == frtright)
            ball_location = lft;
        else if(ball_location == right)
            ball_location = frtleft;
        else if(ball_location == bckright)
            ball_location = frt;
        else if(ball_location == bck)
            ball_location = frtright;
        else if(ball_location == bckleft)
            ball_location = right;
        else if(ball_location == lft)
            ball_location = bckright;
        else if(ball_location == frtleft)
            ball_location = bck;
    }
    else if(change == 180){

```



```

        if(ball_location == frt)
            ball_location = bck;
        else if(ball_location == frtright)
            ball_location = bckleft;
        else if(ball_location == right)
            ball_location = lft;
        else if(ball_location == bckright)
            ball_location = ftrleft;
        else if(ball_location == bck)
            ball_location = frt;
        else if(ball_location == bckleft)
            ball_location = frtright;
        else if(ball_location == lft)
            ball_location = right;
        else if(ball_location == ftrleft)
            ball_location = bckright;
    }
}
else if(direction == LEFT){
    if(change == 45){
        if(ball_location == frt)
            ball_location = frtright;
        else if(ball_location == frtright)
            ball_location = right;
        else if(ball_location == right)
            ball_location = bckright;
        else if(ball_location == bckright)
            ball_location = bck;
        else if(ball_location == bck)
            ball_location = bckleft;
        else if(ball_location == bckleft)
            ball_location = lft;
        else if(ball_location == lft)
            ball_location = ftrleft;
        else if(ball_location == ftrleft)
            ball_location = frt;
    }
    else if(change == 90){
        if(ball_location == frt)
            ball_location = right;
        else if(ball_location == frtright)
            ball_location = bckright;
        else if(ball_location == right)
            ball_location = bck;
        else if(ball_location == bckright)
            ball_location = bckleft;
    }
}

```



```

        else if(ball_location == bck)
            ball_location = lft;
        else if(ball_location == bckleft)
            ball_location = frtleft;
        else if(ball_location == lft)
            ball_location = frt;
        else if(ball_location == frtleft)
            ball_location = frtright;
    }
    else if(change == 135){
        if(ball_location == frt)
            ball_location = bckright;
        else if(ball_location == frtright)
            ball_location = bck;
        else if(ball_location == right)
            ball_location = bckleft;
        else if(ball_location == bckright)
            ball_location = lft;
        else if(ball_location == bck)
            ball_location = frtleft;
        else if(ball_location == bckleft)
            ball_location = frt;
        else if(ball_location == lft)
            ball_location = frtright;
        else if(ball_location == frtleft)
            ball_location = right;
    }
}
}

```

//This function works identically like the update_ball_position function, except
 //this function is used to update and maintain the correct location of the goal
 //in relation to the robot. This then enables the robot to accurately find the goal.

```

void update_goal_position(turn direction, int change){
    if(direction == RIGHT){
        if(change == 45){
            if(goal_location == front)
                goal_location = frontleft;
            else if(goal_location == frontright)
                goal_location = front;
            else if(goal_location == right)
                goal_location = frontright;
            else if(goal_location == backright)
                goal_location = right;
            else if(goal_location == back)
                goal_location = backright;
        }
    }
}

```



```

        else if(goal_location == backleft)
            goal_location = back;
        else if(goal_location == left)
            goal_location = backleft;
        else if(goal_location == frontleft)
            goal_location = left;
    }
    else if(change == 90){
        if(goal_location == front)
            goal_location = left;
        else if(goal_location == frontright)
            goal_location = frontleft;
        else if(goal_location == right)
            goal_location = front;
        else if(goal_location == backright)
            goal_location = frontright;
        else if(goal_location == back)
            goal_location = right;
        else if(goal_location == backleft)
            goal_location = backright;
        else if(goal_location == left)
            goal_location = back;
        else if(goal_location == frontleft)
            goal_location = backleft;
    }
    else if(change == 135){
        if(goal_location == front)
            goal_location = backleft;
        else if(goal_location == frontright)
            goal_location = left;
        else if(goal_location == right)
            goal_location = frontleft;
        else if(goal_location == backright)
            goal_location = front;
        else if(goal_location == back)
            goal_location = frontright;
        else if(goal_location == backleft)
            goal_location = right;
        else if(goal_location == left)
            goal_location = backright;
        else if(goal_location == frontleft)
            goal_location = back;
    }
    else if(change == 180){
        if(goal_location == front)
            goal_location = back;
    }

```



```

        else if(goal_location == frontright)
            goal_location = backleft;
        else if(goal_location == right)
            goal_location = left;
        else if(goal_location == backright)
            goal_location = frontleft;
        else if(goal_location == back)
            goal_location = front;
        else if(goal_location == backleft)
            goal_location = frontright;
        else if(goal_location == left)
            goal_location = right;
        else if(goal_location == frontleft)
            goal_location = backright;
    }
}
else if(direction == LEFT){
    if(change == 45){
        if(goal_location == front)
            goal_location = frontright;
        else if(goal_location == frontright)
            goal_location = right;
        else if(goal_location == right)
            goal_location = backright;
        else if(goal_location == backright)
            goal_location = back;
        else if(goal_location == back)
            goal_location = backleft;
        else if(goal_location == backleft)
            goal_location = left;
        else if(goal_location == left)
            goal_location = frontleft;
        else if(goal_location == frontleft)
            goal_location = front;
    }
    else if(change == 90){
        if(goal_location == front)
            goal_location = right;
        else if(goal_location == frontright)
            goal_location = backright;
        else if(goal_location == right)
            goal_location = back;
        else if(goal_location == backright)
            goal_location = backleft;
        else if(goal_location == back)
            goal_location = left;
    }
}

```



```

        else if(goal_location == backleft)
            goal_location = frontleft;
        else if(goal_location == left)
            goal_location = front;
        else if(goal_location == frontleft)
            goal_location = frontright;
    }
    else if(change == 135){
        if(goal_location == front)
            goal_location = backright;
        else if(goal_location == frontright)
            goal_location = back;
        else if(goal_location == right)
            goal_location = backleft;
        else if(goal_location == backright)
            goal_location = left;
        else if(goal_location == back)
            goal_location = frontleft;
        else if(goal_location == backleft)
            goal_location = front;
        else if(goal_location == left)
            goal_location = frontright;
        else if(goal_location == frontleft)
            goal_location = right;
    }
}

//*****
//ACTIONS
//*****
//*****
//FIND BALL
//*****
//this function is the second function called in order to find the ball. It follows immediately after
//and is
//called by the where_is_ball function which aligns the robot roughly with where it remembers
//the ball to be.
//this function then scans the region in front of it where the ball should approximately be in order
//to find
//it exactly and get possession of it.
void ball_search(){
    int max_light_reading;
    int scope_of_search;
    int amount_searched;
    int num_45deg_turns;
    int milliseconds_scanned;

```



```
int good_enough_light_reading;
```

```
//while we don't have the ball in our possession
```

```
//Technical: LIGHT_2 is the ball sensor
```

```
while(LIGHT_2 < FOUND_BALL_THRESHOLD){
```

```
    cputs("test");
```

```
    msleep(500);
```

```
    //since we don't have the ball, we will look for the maximum light reading we can
```

```
    //find until we find a reading that is high enough to be the ball.
```

```
    max_light_reading = 0;
```

```
    //initially the scope will be one 45 degree angle. It will increase by 45 degrees
```

```
    //each time it fails to find the ball.
```

```
    scope_of_search = 1;
```

```
    //in case the grabbing arm is already down, we need to release it in order to grab
```

```
    //the ball this time.
```

```
    release_grip();
```

```
    //while we don't see the ball either
```

```
    while(max_light_reading < SEE_BALL_THRESHOLD){
```

```
        amount_searched = 0;
```

```
        //spin right at 45 degree increments with increasing scope as we search
```

```
        //updating the goal location as we turn
```

```
        while(amount_searched < scope_of_search){
```

```
            spin_right(time/8);
```

```
            update_goal_position(RIGHT, 45);
```

```
            amount_searched++;
```

```
        }
```

```
        //increase the scope of search for spinning back left so that we stay
```

centered on

```
        //scanning the area we were originally pointed toward but increase our
```

range from

```
        //side to side
```

```
        scope_of_search++;
```

used

```
        //this variable is only necessary for the updating of the goal position. It is
```

when we have

```
        //because we are turning in millisecond intervals, and we have to know
```



```

goal position    //turned enough intervals to add up to a single 45 degree turn. Then the
to know          //will be updated properly, and this variable will be incremented in order
                 //when we have reached the next multiple of 45 degrees. i.e. 90, 135, etc.
                 num_45deg_turns = 1;

                 //keeps track of how many millisecond spins have been executed
                 milliseconds_scanned = 0;

and              //spin back left one millisecond at a time finding the maximum light value

                 //updating the goal location at every 45 degrees
                 while(milliseconds_scanned < (scope_of_search * (time/8))){
                     spin_left(1);

maximum light    //Here is where we actually take readings and attempt to find the
right so that    //we can see, the previous scan only positioned the robot to the

                 //when we scan back left here we are scanning the proper area.
                 //Technical: Once again, LIGHT_2 is the ball sensor.
                 if(LIGHT_2 > max_light_reading)
                     max_light_reading = LIGHT_2;

turn has been    //this if statement updates the goal location each time a 45 degree

                 //made as described above about the variable num_45deg_turns.
                 if(milliseconds_scanned/num_45deg_turns == time/8){
                     update_goal_position(LEFT, 45);
                     num_45deg_turns++;
                 }
                 //increment the millisecond count
                 milliseconds_scanned++;
                 }

//increase the scope of the search in case we didn't find the ball on the last pass
scope_of_search++;
}

//we saw the ball on the last pass
cputs("find");
num_45deg_turns = 1;
milliseconds_scanned = 0;

```



```

//we use a "good enough" light reading because of the real world variance in
readings    good_enough_light_reading = max_light_reading - 5;

//while we don't see the ball as we turn back right to where we saw it earlier
//keep spinning right one millisecond at a time and updating the goal location
every 45 degrees    while(LIGHT_2 < good_enough_light_reading){
                    spin_right(1);

                    //just as above, we have to update the goal location at every 45 degree
interval    //if(milliseconds_scanned/num_45deg_turns == time/8){
            if(milliseconds_scanned/num_45deg_turns > time/16){
                update_goal_position(RIGHT, 45);
                num_45deg_turns++;
            }
            milliseconds_scanned++;
        }

        //now we are directed right at the ball so go toward it as long as it is still in view
        //stop once we find the ball or we don't see it anymore
        while(LIGHT_2 >= good_enough_light_reading && LIGHT_2 <
FOUND_BALL_THRESHOLD){
            go_forward(500);
        }
    }

    //if we have found the ball, stop moving and grab the ball
    stop_motors(1);
    grab_ball();
    ball_location = frt;
}

//this function is the first function called in order to find the ball. It positions the robot facing
//in the general direction of the ball based on where it remembers the ball to be. Then it calls the
//ball search function to narrow in and find the ball precisely.
void where_is_ball(){

    if(ball_location == frt){
    }
    else if(ball_location == bck){
        //turn to face the ball
        spin_right(time/2);
        update_goal_position(RIGHT, 180);
    }
}

```



```

else if(ball_location == lft){
    //turn to face the ball
    spin_left(time/4);
    update_goal_position(LEFT, 90);
}
else if(ball_location == rght){
    //turn to face the ball
    spin_right(time/4);
    update_goal_position(RIGHT, 90);
}
else if(ball_location == frtright){
    //turn to face the ball
    spin_right(time/8);
    update_goal_position(RIGHT, 45);
}
else if(ball_location == frtleft){
    //turn to face the ball
    spin_left(time/8);
    update_goal_position(LEFT, 45);
}
else if(ball_location == bckright){
    //turn to face the ball
    spin_right(3*(time/8));
    update_goal_position(RIGHT, 135);
}
else if(ball_location == bckleft){
    //turn to face the ball
    spin_left(3*(time/8));
    update_goal_position(LEFT, 135);
}
stop_motors(500);
ball_search();
}

//*****
//FIND GOAL
//*****
//this function is the second function called in order to find the goal. It follows immediately after
and is
//called by the where_is_goal function which aligns the robot roughly with where it remembers
the goal to be.
//this function then scans the region in front of it where the goal should approximately be in
order to find
//it exactly.
void goal_search(){
    int max_light_reading;

```



```

int scope_of_search;
int amount_searched;
int num_45deg_turns;
int milliseconds_scanned;
int good_enough_light_reading;

light //we will start with not knowing where the goal is. Then we will look for the maximum
//reading we can find until we find a reading that is high enough to be the goal.
max_light_reading = 0;

//initially the scope will be one 45 degree angle. It will increase by 45 degrees
//each time it fails to find the goal.
scope_of_search = 1;

//while we don't see the goal
while(max_light_reading < SEE_GOAL_THRESHOLD){
    amount_searched = 0;

    //spin right at 45 degree increments with increasing scope as we search
    //updating the ball location as we turn
    while(amount_searched < scope_of_search){
        spin_right(time/8);
        update_ball_position(RIGHT, 45);
        amount_searched++;
    }

    //increase the scope of search for spinning back left so that we stay centered on
    //scanning the area we were originally pointed toward but increase our range from
    //side to side
    scope_of_search++;

    //this variable is only necessary for the updating of the ball position. It is used
    //because we are turning in millisecond intervals, and we have to know when we
    have //turned enough intervals to add up to a single 45 degree turn. Then the ball
    position //will be updated properly, and this variable will be incremented in order to know
    //when we have reached the next multiple of 45 degrees. i.e. 90, 135, etc.
    num_45deg_turns = 1;

    //keeps track of how many millisecond spins have been executed
    milliseconds_scanned = 0;

```



```

right          //spin back left one millisecond at a time for twice the amount of time we spinned
degrees        //finding the maximum light value and updating the ball location at every 45
degrees        while(milliseconds_scanned < (scope_of_search * (time/8))){
                spin_left(1);

                //Here is where we actually take readings and attempt to find the
maximum light  //we can see, the previous scan only positioned the robot to the right so
that           //when we scan back left here we are scanning the proper area.
               //Technical: LIGHT_3 is the light sensor on the top of the robot that sees
the            //light over the goal.
               if(LIGHT_3 > max_light_reading)
                  max_light_reading = LIGHT_3;

               //this if statement updates the ball location each time a 45 degree turn has
been           //made as described above about the variable num_45deg_turns.
               if(milliseconds_scanned/num_45deg_turns == time/8){
                  update_ball_position(LEFT, 45);
                  num_45deg_turns++;
               }

               milliseconds_scanned++;
            }
//increase the scope of the search in case we did not find the goal this time through
scope_of_search++;
}

//we saw the goal on the last pass
cputs("find");
num_45deg_turns = 1;
milliseconds_scanned = 0;

//we use a "good enough" light reading because of the real world variance in readings
good_enough_light_reading = max_light_reading - 5;

//while we don't see the goal as we turn back right to where we saw it earlier
//keep spinning right one millisecond at a time and updating the ball location every 45
degrees        //Technical: LIGHT_3 is the light on top of the robot that can see the lights on the goals.
               while(LIGHT_3 < good_enough_light_reading){
                  spin_right(1);

```



```

        //if(millisecnds_scanned/num_45deg_turns == time/8){
        if(millisecnds_scanned/num_45deg_turns > time/16){
            update_ball_position(RIGHT, 45);
            num_45deg_turns++;
        }
        millisecnds_scanned++;
    }

```

```

//the goal is straight in front of the robot.
stop_motors(1);
}

```

```

//this function positions the robot in the general location of the goal
//based on where it remembers the goal to be

```

```

void where_is_goal(int search_type){
    if(goal_location == front){
    }
    else if(goal_location == back){
        //turn to face the goal
        spin_right(time/2);
        update_ball_position(RIGHT, 180);
    }
    else if(goal_location == left){
        //turn to face the goal
        spin_left(time/4);
        update_ball_position(LEFT, 90);
    }
    else if(goal_location == right){
        //turn to face the goal
        spin_right(time/4);
        update_ball_position(RIGHT, 90);
    }
    else if(goal_location == frontright){
        //turn to face the goal
        spin_right(time/8);
        update_ball_position(RIGHT, 45);
    }
    else if(goal_location == frontleft){
        //turn to face the goal
        spin_left(time/8);
        update_ball_position(LEFT, 45);
    }
    else if(goal_location == backright){
        //turn to face the goal
        spin_right(3*(time/8));
    }
}

```



```

        update_ball_position(RIGHT, 135);
    }
    else if(goal_location == backleft){
        //turn to face the goal
        spin_left(3*(time/8));
        update_ball_position(LEFT, 135);
    }
    stop_motors(500);
    goal_location = front;

    //the more precise searching is done for the offensive search
    //and not for the defensive search
    if(search_type == 1){
        goal_search();
    }
}

//*****
//DRIBBLE or SHOOT
//*****
//this function is used for both dribbling and shooting
//the difference is a different speed is passed in
//in general, this function swings the kicking device
//back, then kicks through, and finally returns the leg back to the
//starting position
void kick(int speed){
    motor_b_speed(speed);

    //move back to kick
    motor_b_dir(rev);
    msleep(500);

    //kick the ball
    motor_b_dir(fwd);
    msleep(400);

    //get back in position
    motor_b_dir(rev);
    msleep(500);

    //stop
    motor_b_speed(0);
    motor_b_dir(brake);
}
//*****
//IMPORTANT SECONDARY FUNCTIONS

```



```

/*****
//This function is used to find the time
//for the robot to spin 360 degrees
//we let it spin twice around to reduce the error introduced
int spin(){
    wakeup_t dummy;
    int start_time;
    int stop_time;

    //take the initial reading of the light intensity to later
    //be used to know when the light is seen
    spin_threshold = LIGHT_3;
    lcd_int(spin_threshold);
    msleep(500);

    //take the start time of the spin
    start_time = sys_time;

    //do the spin
    motor_a_speed(150);
    motor_c_speed(150);
    motor_a_dir(rev);
    motor_c_dir(fwd);
    sleep(2);

    //wait for the robot to see the light which will let it know
    //when it has turned the full 360 degrees
    wait_event(see_light, dummy);

    //take the stop time of the spin
    stop_time = sys_time;

    //stop the spinning
    stop_motors(200);

    //the time for a 360 spin is the stop time minus the start time
    //we divide by 2 because we allow the robot to make two 360 degree rotations
    //the reasoning behind this is that the value saved in "time" will be more
    //accurate if the error in precision is divided across two spins
    time = (stop_time - start_time)/2;
return 0;
}
/*****
int send_data(){

    // parts of this code are copied from the thread_test program in

```



```

// brickOS/util/lnp_tools/

unsigned char listen_addr = 160, brick_addr = 64;
int port = 5;
int i,j;
//char buffer [] = "    this is a test message.\n";
int testing;
//int result;
//int matrix[3][5];

//for (i=0; i<3; i++)
// for (j=0;j<5; j++)
//  matrix[i][j] = i*j;

testing = -625;

// that should be host address 4 for the RCX
lnp_set_hostaddr(brick_addr);

// assume host address 10 for the PC
// common communication port: 5
lnp_reliable_init (listen_addr, port);

//for ( i=0; i<NUM_STATES; i++) {
//    //for( j=0; j<NUM_ACTIONS; j++){
//        //buffer[i] = ' ';
//        //buffer[i+1] = 'x';
//        cputs ("send");
//        msleep(500);
//        //lnp_reliable_write (buffer, sizeof(buffer));
//        //result = lnp_reliable_write ( (char *) &testing, sizeof (testing));
//        //lnp_reliable_printf ("%d %d %d\n", Q_scores[i][j]);
//        //Q_scores[i][j]);//, matrix[1][i], matrix[2][i]);
//        //cputs("  ");
//        //lcd_int (i);
//        //msleep(500);
//        //lcd_int (result);
//        //sleep(2);
//        //}
// }

for (i=0; i<30; i++){
    lnp_reliable_printf ("%d\n ", action_array[i]);
}

```



```

        for (j=0; j<NUM_STATES; j++){
            lnp_reliable_printf ("%X %X %X %X %X\n ", Q_scores[j][0], Q_scores[j][1],
            Q_scores[j][2], Q_scores[j][3], Q_scores[j][4]);
        }

        lnp_reliable_shutdown();
        return 0;
    }
    /*******
int get_reward(int argc, char* argv[]){
    wakeup_t dummy;
    while(1){
        wait_event(touched, dummy);
        reward = 100;
    }
    return 0;
}

    /*******
void what_k(){
    int i;
    int j;
    int value;
    int min, max;
    max = 0;
    min = visits[0][0];

    //find the maximum and minimum values in the visits array
    for(i = 0; i<NUM_STATES; i++){
        for(j = 0; j<NUM_ACTIONS; j++){
            if(visits[i][j] > max)
                max = visits[i][j];
            if(visits[i][j] < min)
                min = visits[i][j];
        }
    }

    //value is used to determine how far along the learning is
    //1 is added to the denominator in case the minimum is zero
    value = max / (min + 1);

    //change the value of k according to how much exploitation of experimentation we need
    if(value < max && value > ((3*max + 1)/4))
        k = 2;
    else if(value < ((3*max + 1)/4) && value > ((max + 1)/2))
        k = 3;
    else if(value < ((max + 1)/2) && value > ((max + 3)/4))

```



```

        k = 4;
    else if(value < ((max + 3)/4) && value > 1)
        k = 5;
    else
        cputs("err");
}

//*****
//INITIATION OF EACH ACTION
//*****
void find_ball(){
    where_is_ball();
}

void off_find_goal(){
    where_is_goal(1);
}

void def_find_goal(){
    where_is_goal(0);
}

void dribble(){
    kick(DRIBBLESPEED);
}

void shoot(){
    kick(SHOOTSPEED);
}

//*****
//FUNCTIONS INVOLVED IN LEARNING
//*****
//function selects the next action to be executed
void select_action(){
    int i;
    float base_value = 0;          //value used to keep the base of where the percent of
                                   //values between 0 and 1 start for the current
    action                         //being observed
    float top_value = 0;           //value used to see if we have reached and encompassed the
                                   //random value
    float random_value = gen_random(); //random value used to pick the action
                                   //must be between 0 and 1
    float action_percent = 0;      //the percent of values associated with the current action

```



```

//being observed
float numerator;          //the numerator of the fraction from the exploitation vs.
                           //experimentation equation
float denominator;        //the denominator for the equation
action_selected = -1;      //the action that will be returned to execute
                           //initialized to -1 so that it can be set to 0 the
first
                           //iteration through the loop
//function determines what the value of k should be
what_k();

//while we have not found the action whose values between 0 and 1 encompass
//the random value, defining the action we will perform, continue
while(top_value <= random_value && top_value != 1){

    //update the action to the next action and the base value to the new base
    action_selected++;
    base_value += action_percent;
    denominator = 0;

    //perform the calculation of the percent for each action
    numerator = power(k, Q_scores[current_state][action_selected]);
    for(i = 0; i < NUM_ACTIONS; i++)
        denominator += power(k, Q_scores[current_state][i]);
    action_percent = numerator/denominator;

    //set the top value to see if the action we just observed is the one to be executed
    top_value = base_value + action_percent;
}
}

//*****
//function performs the action selected
void perform_action(){
    if(action_selected == 0)
        find_ball();
    else if(action_selected == 1)
        off_find_goal();
    else if(action_selected == 2)
        def_find_goal();
    else if(action_selected == 3)
        dribble();
    else if(action_selected == 4)
        shoot();
    else{
        cputs("err");
    }
}

```



```

        msleep(500);
    }
}

//*****
void observe_state(){
    cputs("obs");
    msleep(500);

    if(LIGHT_1 > 31 && LIGHT_1 < 36){
        robot_location = offensive_end;    //white
        cputs("white");
        msleep(500);
    }
    else if(LIGHT_1 > 26 && LIGHT_1 < 31){
        robot_location = defensive_goal;    //red
        cputs("red");
        msleep(500);
    }
    else if(LIGHT_1 > 21 && LIGHT_1 < 26){
        robot_location = offensive_goal;    //beige
        cputs("beige");
        msleep(500);
    }
    else if(LIGHT_1 > 15 && LIGHT_1 < 21){
        robot_location = defensive_end;    //black
        cputs("black");
        msleep(500);
    }
    else
        cputs("err");

    //update the new state after observing the conditions
    new_state = state_array[goal_location][robot_location];
}

//*****
//function updates the q_scores matrix
void update_q_scores(){
    int i;
    float max_k;
    float a;

    //update the visits array to reflect the most recent visit
    visits[current_state][action_selected]++;
}

```



```

//define the value of 'a' used in the updating equation for the weighted average
a = 1.0/(1 + visits[current_state][action_selected]);

//finds the max reward that can be received from our new state
max_k = 0;
for(i = 0; i<NUM_ACTIONS; i++){
    if(Q_scores[new_state][i] > max_k)
        max_k = Q_scores[new_state][i];
}

//the update equation
//Q_scores[0][0] = (1.0 - .5) * Q_scores[1][1] + .5 * (100 + 3);
Q_scores[current_state][action_selected] = (1.0 - a) *
Q_scores[current_state][action_selected] + a * (Y * reward + max_k);
}

//*****
//MAIN PROGRAM
//*****
int main(int argc, char** argv){

    //initializations
    int count = 0;
    bool play = false;
    srand(100);

    ds_active(&SENSOR_1); //sense the ground
    ds_active(&SENSOR_2); //sense the ball
    ds_active(&SENSOR_3); //sense the goal

    //perform spin to find timing for rotations
    spin();

    //this block determines whether we are running it to play or running it to train
    //if we are training then we need the robot to give us time after it performs an action
    //for us to give it a reward, if it is playing, we don't need those hesitations
    //if we hit the touch sensor now, it will play and not train
    reward = 0;
    reward_thread = execi(&get_reward, 0, 0, PRIO_NORMAL,
DEFAULT_STACK_SIZE);
    cputs("rrrr");
    sleep(2);
    if(reward == 100)
        play = true;
    kill(reward_thread);

```



```

//set starting points
goal_location = right;
robot_location = offensive_end;
ball_location = frt;
current_state = state_array[goal_location][robot_location];

//start the reward thread that waits for us to give a reward
reward = 0;
reward_thread = execi(&get_reward, 0, 0, PRIO_NORMAL,
DEFAULT_STACK_SIZE);

//*****
//THE LOOPING LEARNING CYCLE
//*****
count = 0;
while(count < 30){
    cputs("start");
    msleep(500);

    select_action();

    //save the number of the action chosen for later examination
    action_array[count] = action_selected;
    lcd_int(action_selected);
    msleep(500);

    perform_action();

    observe_state();

    //if the robot is training, then it should ask and wait for a reward
    if(play == false){
        cputs("give");
        sleep(3);
        cputs("rrrr");
        msleep(500);
        lcd_int(reward);
        msleep(500);
    }

    update_q_scores();

    //reset variables and update information for next loop
    reward = 0;
    current_state = new_state;

```



```
        count++;

        cputs("end");
        msleep(500);
    }
    //now that training is over, send information back to the computer
    dsound_play(song);
    sleep(10);
    send_data();
return 0;
} //end of program
```