

6.836 Final Project

Evolution in the Micro-Sense: An Autonomous Learning Robot

Chuang-Hue Moh (chmoh@mit.edu)

<http://www.pmg.lcs.mit.edu/~chmoh/836-project>

May 15, 2002



Acknowledgements

I would like to express my appreciation to Dah-Yoh Lim for his advice on machine learning and reinforcement learning techniques, and for kindly reviewing this report. I would also like to thank Ben Leong for his moral support during the project presentation. Last but not least, I would also like to thank Indraneel Chakraborty and Ji Li for their companionship and stimulating challenges that spurred me to have a better understanding of building a real robot.

Abstract

Autonomous learning robots have the advantage over manually programmed robots in that they are able to adapt to varying conditions, both internal to the robot (e.g., energy levels) as well as external environmental conditions (e.g., light, friction). In this project, we implemented a robot that learns how to avoid obstacles using online self-adaptation. We further enhanced the robot's learning ability by incorporating a light-seeking behavior in the robot when its internal energy level is low. This light-seeking behavior uses a tree-like data structure that is based on the concept of eligibility traces used in Q-learning algorithms (reinforcement learning). Finally, we implemented a genetic algorithm on the robot and experimented with using a genetic algorithm as a form of robot learning. The robot was built using the Lego RCX microcomputer and was designed with the subsumption architecture.

Note: Project Web site is at <http://www.pmg.lcs.mit.edu/~chmoh/836-project>.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 2 | Related Work | 3 |
| 2.1 | Robotic Control | 3 |
| 2.1.1 | Deliberative Architectures | 3 |
| 2.1.2 | Reactive Architectures | 3 |
| 2.1.3 | Behavior-based Architectures | 3 |
| 2.2 | Reinforcement Learning | 4 |
| 2.3 | Genetic Algorithms | 5 |
| 2.4 | Analysis & Design | 5 |
| 2.4.1 | Robot Control | 5 |
| 2.4.2 | Reinforcement Learning | 7 |
| 2.4.3 | Genetic Algorithms | 8 |
| 3 | Robot Implementation | 9 |
| 3.1 | Robot Body | 9 |
| 3.2 | Hardware Platform | 9 |
| 3.3 | Actuators & Sensors | 13 |
| 3.4 | Subsumption Network Design I: A Basic Obstacle Avoidance Exploration Robot | 15 |
| 3.5 | Subsumption Network Design II: Light Seeking Behavior | 20 |
| 3.6 | Evolving The Robot Controller Using Genetic Algorithms | 23 |
| 4 | Experiment & Results | 26 |
| 4.1 | Exploration Robot with Obstacle Avoidance | 26 |
| 4.2 | Light Seeking Controller | 27 |
| 4.3 | Evolving the Robot's Controller | 28 |
| 5 | Conclusion & Future Work | 31 |
| A | NQC Codes for Learning Robot | 37 |
| B | LegOS Codes for Evolving Robot Control | 44 |

List of Figures

| | | |
|----|---|----|
| 1 | Genetic Algorithm | 6 |
| 2 | The Robot | 10 |
| 3 | The MIT Programmable Brick | 10 |
| 4 | The Handy Board | 11 |
| 5 | The Lego RCX™ Microcomputer | 11 |
| 6 | Sensors & Actuators | 12 |
| 7 | Bumpers & Touch Sensors | 14 |
| 8 | The Robot Resolving a Collision | 15 |
| 9 | Proximity Sensor | 16 |
| 10 | Obstacle Avoidance Code (NQC) | 17 |
| 11 | The Robot: Sensors & Actuators | 18 |
| 12 | Subsumption Network Design - A Basic Obstacle Avoidance Exploration Robot | 19 |
| 13 | An Illustration of the Robot's Light Seeking Behavior | 21 |
| 14 | Tree-Like Structure Implementing Light Seeking Behavior Controller | 22 |
| 15 | Modes of Operation in Relation to Energy Levels | 23 |
| 16 | Subsumption Network Design with Light Seeking Behavior | 24 |
| 17 | Time Slicing of Light Sensor Between Proximity and Light Sensing | 24 |
| 18 | Robot Resolving A Right Collision While Turning Right | 27 |
| 19 | Tree-Like Structure Implementing Light Seeking Behavior Controller | 29 |
| 20 | Best Evolved Robot Controller's Approximate Behavior | 33 |

List of Tables

| | | |
|---|--|----|
| 1 | Angle Turned Versus Time of Turning | 13 |
| 2 | Distance Traveled Versus Time of Traveling | 13 |
| 3 | Distance of Obstacle Versus Proximity Sensor Reading | 13 |
| 4 | Input Voltage and Sensor Readings | 14 |
| 5 | Parameters for Genetic Algorithm | 25 |

1 Introduction

Applying learning algorithms such as machine learning and reinforcement learning to real physical robots [20, 16, 15] is an area of active research in embodied intelligence. Autonomous, online learning robots possess the ability to operate in complex, dynamic environments through training, instruction and trial-and-error to improve the robots' connection between its perception and action. These learning techniques are often superior than conventional programming because it is often difficult to design a robot that caters to all the variables in the real physical world. Furthermore, programming a physical robot to work in the real world environment is often a laborious and complex task, and the resultant programs are often full of constants and "magic numbers".

Another application of autonomous learning robots is in collaborative multi-robot environments [7]. It was shown that the complex emergent behaviors of insect colonies [1] such as division of labor and self-organization are results of interaction of individuals with simple behaviors and learning capabilities [5]. Individual autonomous, learning robots forms the basis for collaborative robot research.

In this project, we implemented an autonomous, learning robot that performs exploration of the environment and learns how to avoid obstacle effectively. In other words, we integrated sensor learning with robot control and implemented it on a real robot. The robot's controller is implemented with the subsumption architecture [3] that allows the robot's various behaviors (exploration, obstacle avoidance, collision resolution, light seeking etc.) to interact in a hierarchical manner.

In our implementation of the obstacle avoidance behavior in the robot, the threshold used by the proximity sensor for obstacle detection can vary according to the robot's speed, battery power and other environmental conditions. Through online self-adaptation, the robot will learn to adjust this parameter according to its own physical state and its surrounding environment. Furthermore, we also provided the robot with the additional behavior of seeking light when its internal energy level runs low. The light seeking behavior is implemented using a tree-like data structure that is based on the concept of eligibility traces [14] used in Q-learning [19] algorithms in a non-Markovian environment. This data structure allows the robot to remember the direction of highest light intensity and navigates the robot towards that direction, and fine tunes its direction in every step. The fine-tuning is performed by decreasing the angle of turning at every search step e.g., the robot first performs 90° left/right turns in the first step to search for the highest light intensity level, and 45° in the second step. This behavior can be applied to robots with solar energy panels for it to recharge its batteries or to keep robots within communication range of a base station while the robot is exploring its surrounding environment.

Although self-adaptation and robot learning does provide us with a more robust design, programming a robot controller and fine-tuning its parameters are tedious and time-consuming tasks. The final phase of our project involves the implementation of genetic algorithms to evolve an obstacle avoiding robot. In other words, we utilize genetic algorithms as a form of robotic learning.

The rest of the report is organized as follows. Section 2 summarizes some of the related work in robot controller design, reinforcement learning and genetic algorithms. Section 3 details the robot implementation and Section 4 presents the results and insights we gained by running experiments on the robot. Finally Section 5 conclude this report and presents some areas of future work.

2 Related Work

2.1 Robotic Control

2.1.1 Deliberative Architectures

Deliberative architectures are based on symbolic artificial intelligence, which is in turn, based on the physical symbol system hypothesis [12]. The essence of this hypothesis can be summarized as follows:

A physical symbol system has the necessary and sufficient means for intelligent action

Deliberative control takes into consideration all of the available sensory information and amalgamates them with all the controller's internal "knowledge" to create a plan of action. A symbolic model of the world is explicitly represented and decisions are made based on logical reasoning. The control searches through all the possible actions plans until it finds a suitable one. This search sequence can take a long time and is hence not suitable in situations where the robots are expected to react quickly. Furthermore, there is often a problem in translating the real physical world into an accurate and sufficient symbolic representation for the robot to make meaningful decisions. Prodigy [4] is an example of a deliberative learning architecture.

2.1.2 Reactive Architectures

In sharp contrast to deliberative architectures are reactive architectures, where the perception is tied closely to the effector action. The architecture does not entail any kind of symbolic world model and does not use complex reasoning. The reactive control is essentially a reflex mechanism where stimulus-response pairs govern actions. The main advantage of robots with reactive control is that they respond quickly to a changing environment where no a priori information is available. The system requires small amount of memory and does not compute or store representations of the world. The inability to learn over time is perhaps its main drawback of reactive architectures.

2.1.3 Behavior-based Architectures

A behavior-based architecture is one that is built from a collection of behaviors that achieve or perform certain goals. By combining and controlling these behaviors, complex robotic actions can emerge. This class of architecture originated from the subsumption architecture [3] developed in the mid-1980s. In this architecture, the robot control is built in a bottom-up fashion, with task-achieving actions/behaviors as components. These components can be executed in parallel and newly added components and layers are built on top of existing ones, without modifying the way the existing components behave. The notion of internal model is absent in this architecture as "the world is its own best model". The main limitation of this type of

architecture is that the robot’s goals must be capable of being represented implicitly in the controller’s structure according to a fixed, pre-compiled ranking scheme. The Genghis [2] is a six-legged robot architected using the subsumption model.

2.2 Reinforcement Learning

Reinforcement learning is a trial-and-error learning technique where the agent (robot) is given a scalar reinforcement signal as a response to its actions (but clueless as to what actions would have been best). The agent tries to maximize the reinforcement signals from the environment over time by continually interacting with the environment, with the agent selecting actions and the environment responding to those actions and presenting new situations to the agent.. The reinforcement signals are called *rewards*. In general, the agent will seek to maximize the *expected return* of the rewards, R_t :

$$R_t = r_{t+1} + r_{t+2} + \dots + r_{t+n}$$

where $r_{t+1} + r_{t+2} + \dots + r_{t+n}$ is the sequence of rewards received after time step t . The agent’s task is to find a policy that maps states to actions that will maximize the expected return. The model most widely used in reinforcement learning is the *infinite horizon discounted model*, where the agent tries to maximize the *expected discounted return* of rewards R_{td} :

$$R_{td} = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where $0 \leq \gamma \leq 1$ is called the discount rate. The higher the γ value, the more important future rewards are to the agent. Conversely, a γ value of 0 implies that the agent is only concerned with the immediate return and we called this type of algorithms greedy. The classical problem of *exploitation versus exploration* centers around the choosing of the optimal γ value.

The environment is defined as anything external of the agent i.e., anything that cannot be changed arbitrarily by the agent is considered to be outside of it and thus part of the environment. It is often a mistake to think of the interface between the agent and the environment as the physical boundary of the robot’s body. For example, the motors and mechanical linkages of a robot and its sensing hardware should usually be considered parts of the environment rather than part of the agent.

One branch of reinforcement learning, called the Q-learning technique [19], is widely used in *model-free* methods i.e., learning of a policy without building a model of the environment. In this technique, the algorithms store the expected return associated to each state-action pair. Q-learning is very simple because only the state-action function needs to be stored and updated. The policy greedily select the action that will return the maximum value at each state. Iterative refinement of the state-action’s value is done to make it closer to the “real” return value. More formally, the action-state function, $Q(s, a)$ is defined as:

$$Q(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') \max_{a'} Q(s', a')$$

where $R(s, a)$ denotes the immediate reward of taking actions a in state s and $T(s, a, s')$ is the probability of making the transition from s to s' .

Reinforcement learning is often a considerable abstraction of the problem of goal-directed learning from interaction. The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of reinforcement learning.

A state signal that succeeds in retaining all relevant information that summarizes past sensations compactly while retaining all relevant information is said to be Markov. In other words, the state transitions have to be independent of the path of the agent until this point and only depends on the current state. For example, in a game of chess, the current configuration of all the pieces on the board would serve as a Markov state because it summarizes everything about the complete sequence of positions that led to it although information about the sequence is lost. The sequencing information however, does not affect the future information and hence is insignificant to an agent that learns how to play chess. If an environment has the Markov property, then its one step dynamics enable us to predict the next state and expected next reward given the current state and action. A reinforcement learning task that satisfies the Markov property is called a Markov decision process. The Markov property allows some reinforcement learning algorithms to achieve optimality but it is not always realistic to assume that it holds in real-world situations.

2.3 Genetic Algorithms

Genetic Algorithms (EA) is a class of algorithms that is inspired by the Darwinian theories of evolution and natural selection proposed by British naturalist Charles Darwin in his publication *On the Origin of Species* in 1859. A basic genetic algorithm is outline in Figure 1. As seen from the figure, there are many parameters that can be adjusted to obtain different emergent behavior of the population as a whole. Genetic algorithms provide us with a new programming paradigm for programming robot controller without deliberate design and tedious parameter tuning on the part of the programmer - the controller is simply evolved over time.

2.4 Analysis & Design

2.4.1 Robot Control

Traditional goal-based robot design has been applied successfully to many domains such as software agents and Web crawlers. Deliberative architectures, for example, are more widespread in software agent technologies rather than physical robot control. The main reason for this is that software agents themselves exists in

-
1. Randomly create an initial population of POPULATION_SIZE individuals P_0 .
 2. Repeat the following for MAX_GENERATION steps:
 - (a) Compute and store the fitness for each individual in the current population. Rank the individuals according to their fitness values.
 - (b) Compute $D = \frac{1}{d^2}$ for each individual in the current population, where d is the number of bits (in the genotype) that the individual differs from the genotype of the fittest. Rank the individuals in decreasing d value.
 - (c) Rank the individuals according to the combined rank of the preceding two steps.
 - (d) Repeat POPULATION_SIZE $\times R_{crossover}$ times:
 - i. Pick 2 progenitors by walking down the population rank and choosing an individual with probability p_{choose} given that previous individual has not been chose.
 - ii. Produce a new individual by crossover on the two progenitors and add it to the new generation.
 - (e) Repeat POPULATION_SIZE $\times R_{mutation}$ times:
 - (f) Pick 1 individual by walking down the population rank and choosing with probability of p_{choose} given that previous individual have not be chosen. Produce a new individual through mutation of the chosen individual.
 - (g) Add each individual in the current generation to the new generation.
 - (h) Rank each individual in the new generation using the methods outline above.
 - (i) Choose POPULATION_SIZE robot controllers to survive in the new generation.

Note:

- **Crossover:** Randomly choose a crossover point among all the bits in the genotype. Concatenate the two genotype, using the bits from the start of the first individual's genotype to the crossover point, and the bits from the crossover point to the end of the genotype from of the second individual.
 - **Mutation:** Flip each bit in the genotype with a small, independent probability, p_{mutate} .
-

Figure 1: Genetic Algorithm

a symbolic world that is deterministic, and that time is often not a constraint for these agents. This implies that an accurate model of the world can be provided to the robot agents to formulate their plans and actions and that the agents have the luxury of time to formulate these action plans without having to worry about the time required for reading sensors and controlling actuators.

However, assuming an accurate representation of the real world is often unrealistic, especially in designing physical robots. One reason for this is because of the dynamism of the real world. For example, the brightness of ambient light is hardly an indication of the time of the day, depending on many factors such as weather conditions, seasons etc. With an almost infinite number of parameters to consider, it would be impossible to accurately represent the real world in a way that will allow robots to function deterministically in the traditional goal-based approach. Another reason is due to the fact that sensors are unable to present a complete picture of the environment - the sensors may be not accurate and precise enough or the sensor channel may be noisy.

The speed of reaction of the robot is also important because it is not realistic to assume that there are not costs associated with using sensors and actuators. In a real robot, it takes time for the sensor to provide the microprocessor with readings and for the microcomputer to move the actuators. Therefore, we cannot afford to have the robot controller perform intensive computation before reacting to the stimuli provided to the robot. Although reaction time is important, we believe that designing autonomous robots to do the “right thing” is arduous if there is no notion of memory or learning - something that reactive architectures are deficient in.

In view of these considerations, we have implemented our robot control using the subsumption architecture to provide the robot with acceptable performance when reacting to its environment and the flexibility to learn and evolve the robot controller over time.

2.4.2 Reinforcement Learning

A serious limitation in putting reinforcement learning techniques into physical robot is that Markov property is seldom fulfilled in the real world. When this happens, the theoretical guarantees of convergence to optimal behavior do not apply anymore. The first reason for the failure of the Markov property is due to the fact that the robot’s limited perceptual information prevents it from determine its exact state. This problem is widely known as *hidden state* or *perceptual aliasing*. Secondly, current technologies such as machine vision does not provide noise free sensory information to the robot. This makes the environment non-deterministic because two consecutive readings of the sensor at the same location may not yield the same results.

Another practical problem with reinforcement learning is that it is a memory intensive task. Consequently, it is often infeasible to map all the raw sensor values into the state-space. Some form of state-space pruning, classification or classification must be applied to make reinforcement learning a practically implementable task. These methods can give rise to imprecise definition of states and hence violation of the Markov property. Eligibility traces [14] are applied to Q-learning algorithms in a non-Markovian environment to improve their performance. These traces record the agent's previous experiences and not just the last step.

In this project, we implemented a tree-like data structure that records the robot's experiences during its search for a region of high light intensity. This tree-like data structure, together with the robot's ability to fine-tune its search towards the light source, will enable the robot to zero into the light source over time. This concept of the robot remembering its last few moves and fine-tuning its moves with respect to that memory is similar to that of an eligibility trace mentioned above.

2.4.3 Genetic Algorithms

There are two main obstacles in implementing genetic algorithms in physical robots. Firstly, the programmer needs to know how to define the fitness function used to evaluate the individuals. This is often a deliberate and complicated task. This task is very similar to supervised learning methodologies used in traditional machine learning and neural networks. Secondly, genetic algorithms often require a fairly large number of individuals and many generations of simulation of this population. This makes running genetic algorithms on robot controllers a very time consuming tasks due to the sheer size of number of simulations that needs to be run on the robots. Given the high cost of building robots, using pure genetic algorithms to evolve robot controllers is, more often than not, a prohibitive task e.g., in high risk urban missions. Currently, genetic algorithms are often applied in simulations and the best evolved controller is downloaded into real robots. However, due to the differences between the ideal world in simulation and the non-ideal world in reality, this method does not provide us with a robust controller that will allow the robot to react optimistically in real world environments.

Nonetheless, genetic algorithms provide a way in which complex behaviors can be derived from a group of individual, autonomous robots. It also allows the population as a whole learns the "tricks to life" through evolution. Although it is not clear if genetic algorithms are feasible in the implementation of physical robots, it is still interesting to experiment with these algorithms to derive interesting emergent behavior (if any) from the robots.

3 Robot Implementation

3.1 Robot Body

We made use of Lego parts for our body design. We made use of Legos because it provides a very quick way of prototyping the physical body of the robot and allows us to modify our design with ease. The problem with Lego parts is that they come apart very easily and this is not desirable, especially if the robot is supposed to collide into obstacles (to learn how to avoid them). We interlocked the various parts of the robot's body so that the parts do not fall off easily. This inevitably adds weight to the robot's body and hence reduces its angular velocity and speed.

Finally, we note that by keeping to a minimum configuration of two motors and two wheels connected directly to the motors, the robot can move relative fast. However, this makes the robot movement jerky and horizontally unstable (oscillates horizontally when the robot stops), causing sensor noise in some cases (especially in the light sensor). Therefore, we choose to make the robot's body wider and with four wheels so that it can have better stability and sensor noise can be reduced. In addition, by having a wider distance between the wheels and using gears to connect the wheels and the motors, we reduced the maximum speed of the robot while increasing its torque. This gave us better stability and greater granularity in controlling the robot's speed. Furthermore, we do not want to robot to move too fast since our goal is to have the robot continuously sensing and moving rather than stopping after every step - if the robot moves too fast, the sensors may not be fast enough to provide accurate information and/or sufficient reaction time for the robot.

3.2 Hardware Platform

The robot needs a central processing unit or its "brain" that converts the input signals to digital form, process these digitized input data and controls the actuators i.e., actions of the robot. There are a variety of hardware platforms that are suitable for this purpose. The main considerations in choosing the robot's brain are the number of I/O ports, availability of sensors, programmability and cost. The Motorola 68HC11-based Handy Board [17] and the Hitachi H8/3292-based Lego RCXTM were two likely candidates that provide both the flexibility of the hardware and programmability of the software, both of which are desirable features in this project. Incidentally, the Lego RCX was inspired by the MIT Programmable Brick Project [18] in the MIT Media Lab, while the Handy Board originated from the same research group in the Media Lab. Pictures of the MIT Programmable Brick and the Handy Board are shown in Figures 3 and 4 respectively.

The Handy Board is based on the 8-bit MC68HC11 microprocessor that runs at 3 MHz and has 32KB of battery-backed static RAM. The CPU has a 16-bit address bus and has build-in timers, A/D converters, and

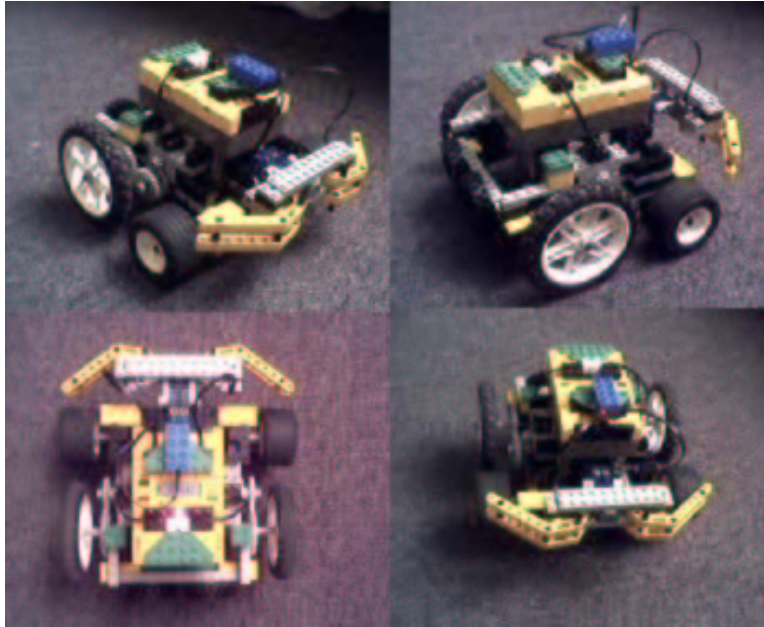


Figure 2: The Robot



Figure 3: The MIT Programmable Brick

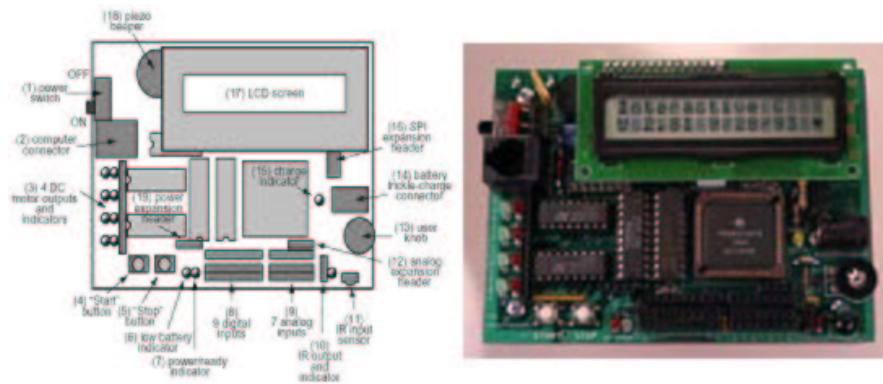


Figure 4: The Handy Board

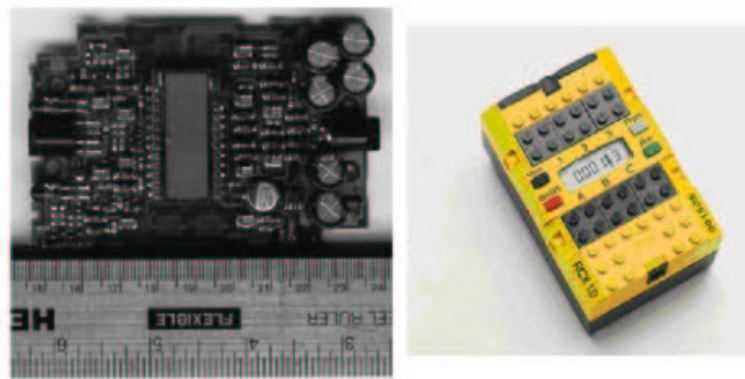


Figure 5: The Lego RCX™ Microcomputer

synchronous (RS232) and asynchronous (SPI) communication channels. In addition, the CPU uses memory-mapped I/O to control the input and output ports. The Handy Board has four outputs for DC motors and a connector system for plugging in active sensors. The programming language used for the Handy Board is Interactive C, which follows closely to the C programming language syntax.

The core of the RCX, on the other hand, is a Hitachi H8/3292 microcontroller with 32K of external RAM and 16K of on-chip ROM. The H8/3292 is a 8-bit general register microcontroller with 16-bit address space and runs at 16 MHz. The RAM is used to store user programs while the ROM is used to store a driver that is run when the RCX is powered up. The driver is extended by downloading 16K of firmware to the RCX.

There are three output ports and three input ports on the RCX. The internal 10-bit A/D converter is used to interface between the CPU and the input ports of the RCX. Each input port is connected to an analog

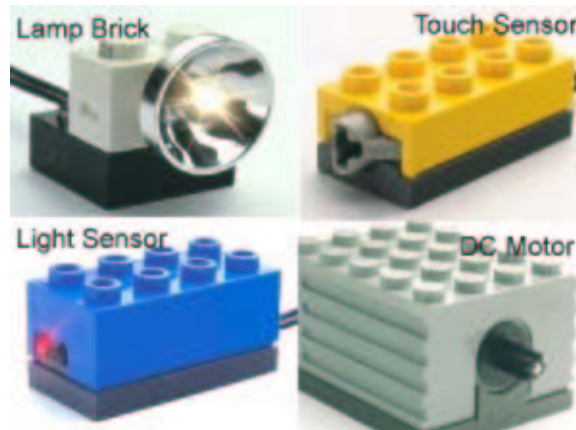


Figure 6: Sensors & Actuators

input pin of the A/D converter. The sensors connected to the input ports are divided into two types - active (light and rotation sensors) and passive (touch and temperature sensors). The sensor input values is obtained through A/D conversion. The device registers of the A/D converted is used to initiate a conversion, to monitor the end of conversion, and to access the resulting converted 10 bit sensor input value. The A/D control/status register is used to start conversion and to poll for the end of conversion. The end of conversion can also be signaled with an A/D end interrupt. The A/D data registers store the result of the conversions. The output ports are driven by current from the 9V battery power supply. The current flowing through the output ports are controlled by writing to the device register at address 0xf000 (each port is controlled by two bits).

Programs are downloaded to the RCX via an infra-red Tower, which is connected to the PC's USB port. The RCX has an in-built infra-red transmitter and receiver for the purpose of communicating with the IR tower and other RCX bricks. The RCX can be programmed using a variety of languages, like the C-like Not-Quite-C (NQC) [13], C/C++ on legOS [9] and Java on leJOS [10].

As the RCX and Handy Board both offers similar functionalities and programming tools for the purpose of this project, we choose to implement the robot's brain using the Lego RCX due to the lower cost. Building the robot using the Handy Board will cost twice as much as using the RCX. Furthermore, the RCX runs at a faster clock rate than the MC68HC11 microcontroller used in the Handy Board. This is more attractive to us because we are running a number of concurrent tasks in the robot's controller.

| | | | | | |
|-------------------------------|-----|-----|------|------|------|
| Angle Turned | 45° | 90° | 180° | 270° | 360° |
| Time of Turning (secs) | 1.0 | 1.9 | 4.1 | 6.1 | 8.2 |

Table 1: Angle Turned Versus Time of Turning

| | | | | | |
|-------------------------------|-----|-----|-----|-----|-----|
| Time of Travel (secs) | 0.5 | 1.0 | 1.5 | 2.0 | 2.5 |
| Distance Traveled (mm) | 132 | 267 | 378 | 516 | 653 |

Table 2: Distance Traveled Versus Time of Traveling

3.3 Actuators & Sensors

In this project, we make use of two 9V DC servo motor to drive the robot. The two motors are connected to the first (A) and third (C) output ports provided in the RCX and the program will write into the registers of the output ports to control the speed of the motor.

We measured the various parameters of the motor using 6 new 9V batteries and they are shown in Tables 1 to 3. As we can see from the values in these tables, the linear and angular distances traveled are proportional to the time that the motor runs (with a small initial start-up cost). The angular velocity is 42.9°/sec and the velocity of the vehicle is 257 mm/sec.

The input voltage and the sensor values are show in Table 4. The input is converted into an internal raw value in the range 0 (0V) to 1023 (5V). The touch sensor values and the light sensor values that are shown in the table are the values reported by RCX to the programmer, depending on whether the programmer defined the input port to be connected to a light sensor or touch sensor.

One end of the touch sensors (simple switch) are connected to the bumpers of the robot and the other end is connected to the input ports (1 and 3) of the RCX to provide collision detection as shown in Figure 7. Whenever the bumper collides with an obstacle, it will cause the touch sensors to be pressed, which in turn generates an interrupt at the microcontroller. The program reacts to this interrupt with a collision resolution behavior that moves the robot away from the obstacle. The bumpers were built in such a way that its width

| | | | | |
|-----------------------------------|-----|-----|-----|-----|
| Distance for Obstacle (mm) | 0 | 127 | 254 | 381 |
| Proximity Sensor Readings | 222 | 128 | 83 | 41 |

Table 3: Distance of Obstacle Versus Proximity Sensor Reading

| Voltage (V) | Raw Value | Light Sensor Value | Touch Sensor Value |
|-------------|-----------|--------------------|--------------------|
| 0.0 | 0 | - | 1 |
| 1.1 | 225 | - | 1 |
| 1.6 | 322 | 100 | 1 |
| 2.2 | 450 | 82 | 1 |
| 2.8 | 565 | 65 | 0 |
| 3.8 | 785 | 34 | 0 |
| 4.6 | 945 | 11 | 0 |
| 5.0 | 1023 | 0 | 0 |

Table 4: Input Voltage and Sensor Readings

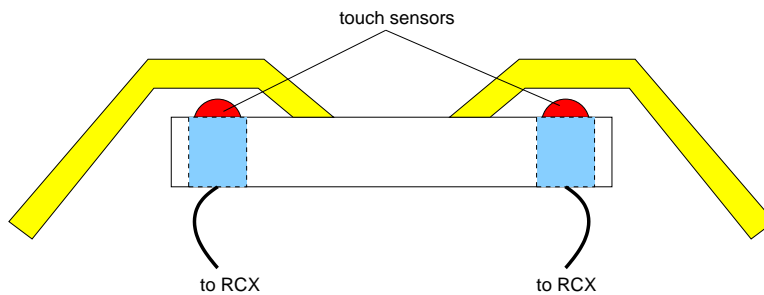


Figure 7: Bumpers & Touch Sensors

is almost as wide as the robot's body so that it can successfully detect any collision when it occurs. An illustration of the robot resolving a collision is shown in Figure 8.

We constructed our proximity sensor by using the combination of the RCX's in-built IR transmitter and the light sensor (0.6 to 760 Lux) as illustrated in Figure 9. The IR transmitter is in-built into the RCX while the light sensor is connected to an input port (2). The IR transmitter will broadcast packets and the light sensor will measure the difference between two consecutive readings to determine if there is an obstacle in front of the robot. The light sensor is very sensitive to infra-red signals, hence when the infra-red packet sent by the IR transmitter gets reflected back by an obstacle in the path of the robot, it will cause a jump in the light sensor's readings. The NQC code for performing this operation is shown in Figure 10. A proximity is detected by an abrupt change to two consecutive light sensor readings, indicating that the IR packets sent out by the `send_signal()` task has been reflected back. We place the light sensor slight above the IR transmitter in our robot so that the outgoing IR packets do not interfere with the readings of the light sensor. We also ensure that the IR transmitter is not blocked by other parts of the robot causing the proximity sensor

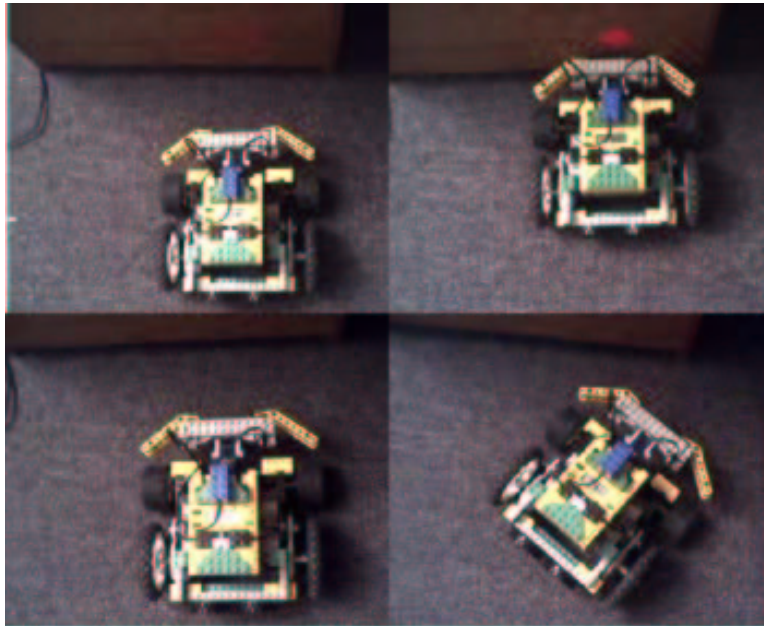


Figure 8: The Robot Resolving a Collision

to give erroneous results.

Through trial-and-error, it was found that the proximity sensor is most effective if the RCX sends out a burst of four consecutive packets before we check for any reflections. However, we keep our implementation to sending one packet at a time as the robot will eventually learn how to avoid obstacles effectively. Furthermore, the robot controller will also learn how to adapt to different environments by adjusting the threshold (difference between two consecutive light sensor readings) for obstacle detection.

Finally, we made use of a light brick as an indicator of the internal energy level of the robot. The light brick is a 9V DC LED that is connected to the second output port (B) of the RCX and is controlled in the same way as the motors i.e., by writing a value to the register of the output port (B). Figure 11 shows how the different sensors and actuators are connected in the robot.

3.4 Subsumption Network Design I: A Basic Obstacle Avoidance Exploration Robot

We first implemented a simple robot using the subsumption network architecture as shown in Figure 12. This robot controller comprises of augmented finite state machines (AFSM) connected in a hierarchical manner. The robot learns how to avoid obstacle using online self-adaptation, which allows the robot to

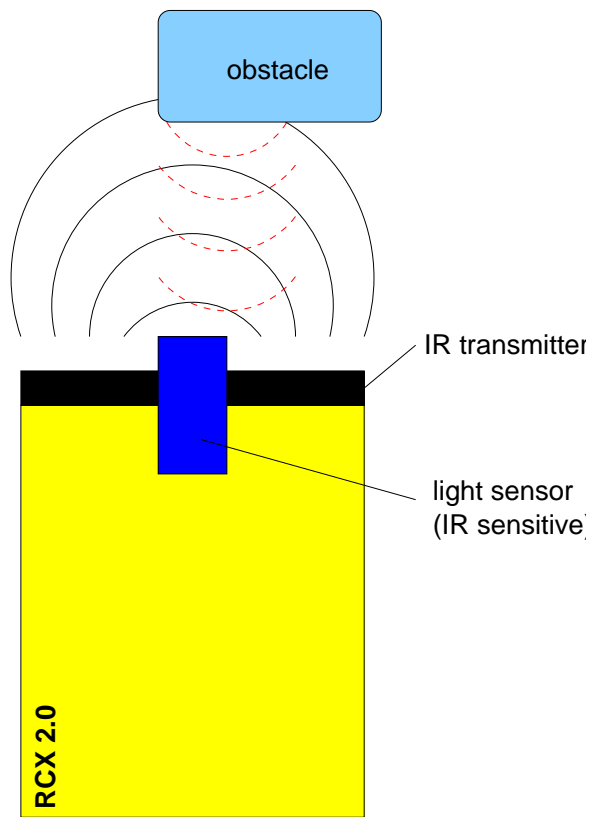


Figure 9: Proximity Sensor

```

task send_signal() {
    SetPriority(MIN_PRIORITY-2);
    while(true) {
        SendMessage(0); /* Sends IR packets, takes approx 40 ms*/
        Wait(10);      /* Wait for 100 ms to allow detection */
    }
}

task avoid_obstacle() {
    SetPriority(MIN_PRIORITY-3);
    int previous_proximity;
    int diff;

    start send_signal;
    while ( true ) {
        /* Calculate the difference between two consecutive
           readings of the light sensor */
        previous_proximity = LIGHT_SENSOR;
        diff = LIGHT_SENSOR - previous_proximity;
        diff = ( diff < 0 ) ? -diff : diff;
        if ( diff > proximity_threshold ) {
            acquire(ACQUIRE_OUT_A + ACQUIRE_OUT_C) {
                move_backward();
                turn_right();
            }
        }
    }
}

```

Figure 10: Obstacle Avoidance Code (NQC)

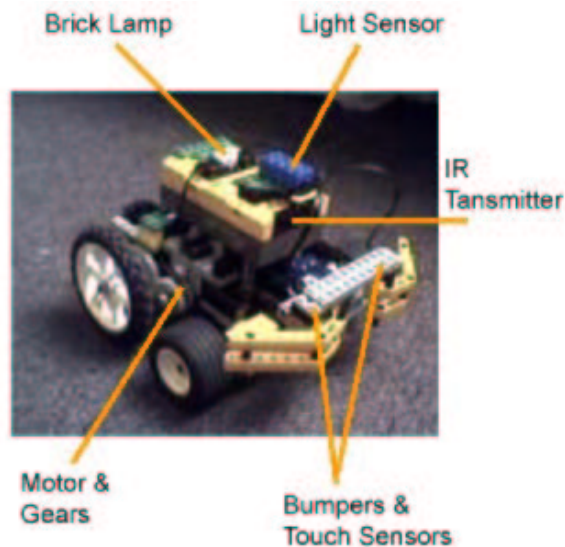


Figure 11: The Robot: Sensors & Actuators

adjust its proximity detection threshold according to its internal energy level and environment conditions such as brightness and friction.

- **Left-Speed/Right-Speed AFSM:** The Left Speed and Right-Speed AFSMs are connected to the left and right motors of the robot respectively to control the motor speed. By varying the left and right motor speeds, the robot can move forward, turn left, turn right and reverse.
- **Move-Forward AFSM:** The Move-Forward AFSM provides equivalent signals to the Left-Speed and Right-Speed AFSMs to cause the robot to move the vehicle forward. Whenever it detects than any of the motors have stopped, it signals the appropriate AFSM (Left-Speed or Right-Speed) to restore the motor speed. There are two ways that the Move-Forward AFSM can maintain the motor speed: keep signaling the Left-Speed and Right-Speed AFSM with the same constant values or through a feedback loop that monitors the motors' speeds.
- **Turn-Left/Turn-Right AFSM:** The Turn-Left and Turn-Right AFSMs will suppress the signals from the Move-Forward AFSM and signal the Right-Speed and Left-Speed AFSMs with negative values (reverse) to turn the robot left and right respectively. Note that in order to turn the robot in one direction, we move one wheel forward and the other wheel backwards. Simply stopping one motor will not allow the robot to turn properly because the force from the moving wheel will cause the stationery wheel to move as well.
- **Explore AFSM:** While the previous AFSMs provide basic control for the robot, the Explore AFSM provides the robot with the ability to explore its environment randomly. This AFSM uses a random

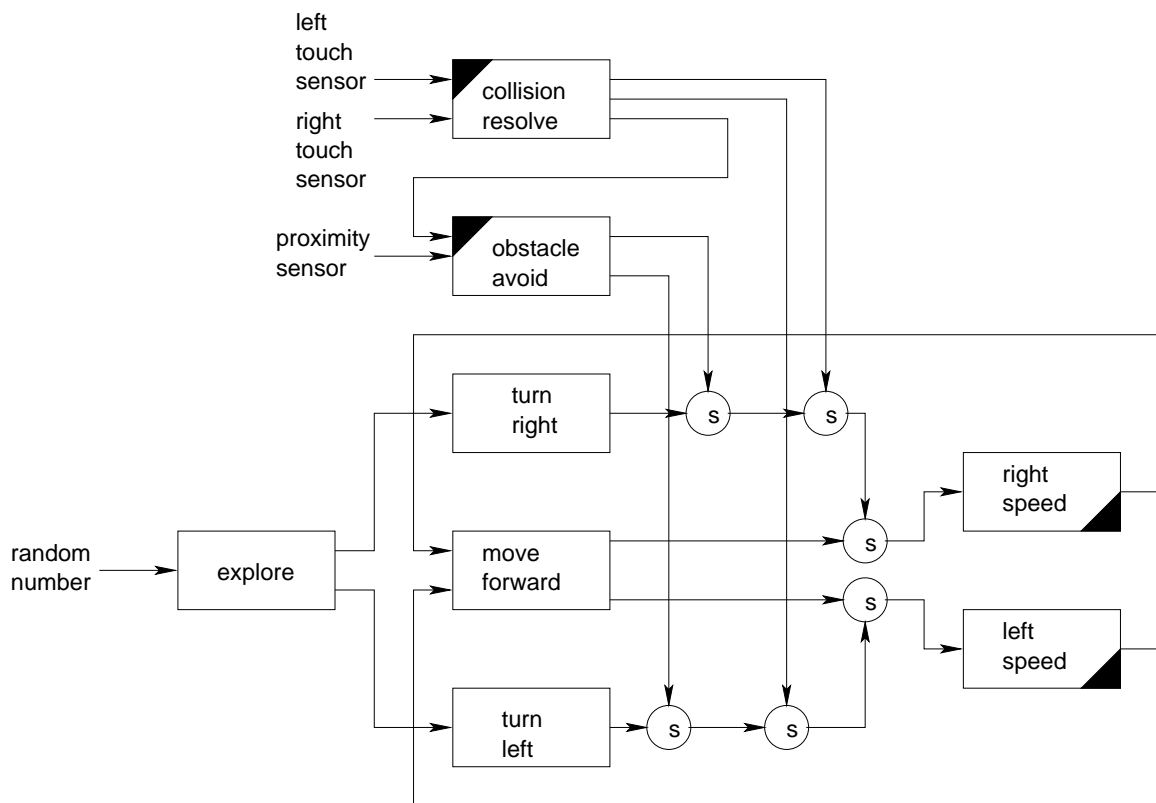


Figure 12: Subsumption Network Design - A Basic Obstacle Avoidance Exploration Robot

number generator to determine the robot's next move. It will then do one of the following actions:

- *More Forward*: No action required as moving forward is the robot's default action as dictated by the Move-Forward AFSM.
- *Turn Left*: Signals to the Turn-Left AFSM to turn the robot left.
- *Turn Right*: Signals to the Turn-Right AFSM to turn the robot right.

Note that the decision of the next step is done in a periodic basis using the AFSM's internal timer.

- **Collision-Resolve AFSM**: The Collision-Resolve AFSM takes inputs from the left and right touch sensors and determines if a collision on the left/right bumper has occurred. It will then attempt to resolve any collisions that has occurred by:

- Signals negative values to the Right-Speed and Left-Speed AFSMs to reverse the robot away from the obstacle.
- Signals the Turn-Right AFSM to turn the robot right if a collision at the left bumper has occurred. Signals the Turn-Left AFSM to turn the robot left if a collision at the right bumper has occurred.

The Collision-Resolve AFSM also feedbacks to the Obstacle-Avoid AFSM to adjust its obstacle detection threshold.

- **Obstacle-Avoid AFSM**: The Obstacle-Avoid AFSM avoids obstacle by reading the values from the proximity sensor. When this value is more than a certain threshold, the AFSM will conclude that there is an obstacle in front of the robot. It will then control the robot to reverse and turn either left or right to avoid the obstacle. The threshold value is initially set of a very large value so that the AFSM will ignore any signals from the proximity detection sensor. The threshold will be adjusted using feedback from the Collision-Resolve AFSM. Eventually, this will eventually allow the robot to adapt and avoid obstacles.

3.5 Subsumption Network Design II: Light Seeking Behavior

The robot controller's state transition for the light seeking behavior is in the form of a tree-like data structure (in reality, it is a graph structure) shown in Figure 14. This structure allows the robot to memorize its recent "experiences" i.e., light sensor values, when turning around seeking a direction of highest light intensity while minimizing the state information kept. At each turn, the robot remembers (using a single variable) the maximum light intensity detected and the compare function returns a + if the current light level is higher (and sets the maximum light level to the current light level). Otherwise, it outputs a -. This output symbol (+/-) is used as inputs to the tree structure in Figure 14. The state transition caused by following the data

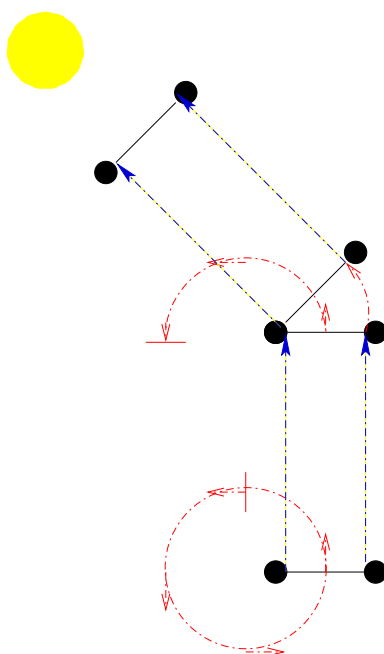


Figure 13: An Illustration of the Robot's Light Seeking Behavior

structure will allow the robot to turn towards the direction of highest light intensity when it completes its initial 360° search. At each iteration of the data structure, the robot will move forward one step and reduce its angle of turning by half. In this way, the robot progressively fine-tunes its position towards the direction of highest intensity. Figure 13 illustrates this fine-tuning behavior. We could see that this data structure is very similar in concept to eligibility traces used in Q-learning algorithms in that it allow the robot to memorize a trace of its previous moves.

As we have mentioned earlier, the robot operates in two modes, namely the explore mode and the recharge mode. These modes are determined by its internal energy level as shown in Figure 15. We make use of threshold levels to determine when a robot should switch mode. When the robot's energy decreases from a maximum of 400 and reaches the lower threshold of 150, it switches from explore mode to recharge mode and starts to seek light. Conversely, when the robot's energy increased from 0 to 250, it is in the recharge mode. The robot jumps back to the explore mode when the energy level raises above the higher threshold value of 250. The use of two threshold values instead of one to switch mode prevents the robot from getting trapped in the situation where it keeps swinging between two modes. Note that in our design, one move of the robot's single motor consumes 1 unit of internal energy. When the light level of the surrounding environment reaches above a `RECHARGE_THRESHOLD` (perceived by the light sensor), the robot gains internal energy, which is the difference between the `RECHARGE_THRESHOLD` and the light reading of

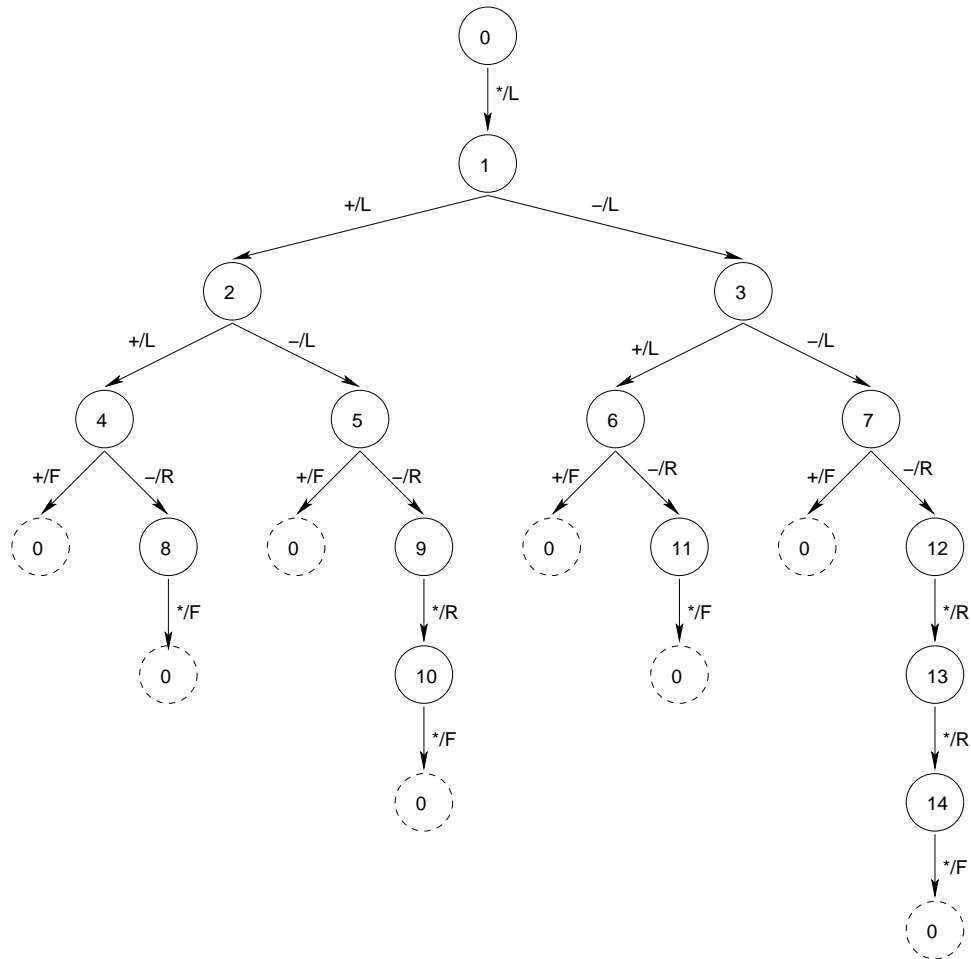


Figure 14: Tree-Like Structure Implementing Light Seeking Behavior Controller

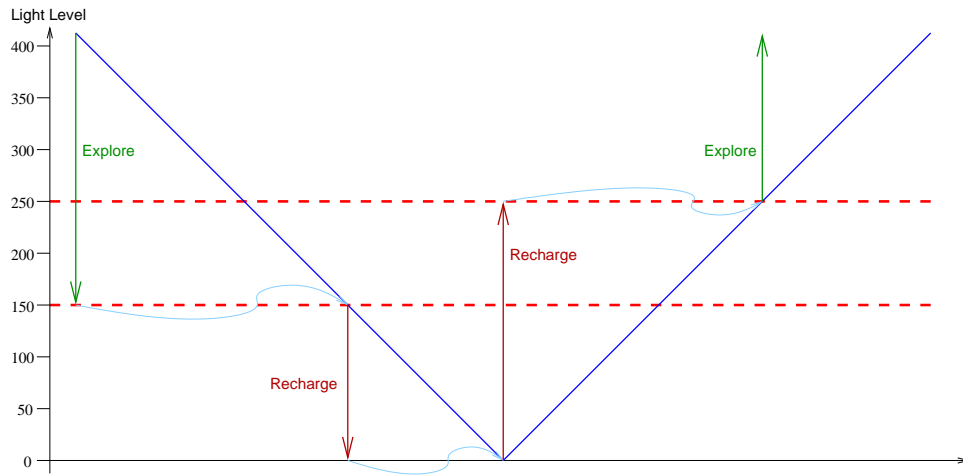


Figure 15: Modes of Operation in Relation to Energy Levels

the sensor.

The modified subsumption network architecture is shown in Figure 16. The main difference between this modified network and the original subsumption network shown in Figure 12 is the addition of the **Recharge** AFSM that subsumes the output of the explore AFSM. This AFSM takes the light level (from the light sensor) and the energy level (from the internal energy sensor) as inputs and determines if the energy level of the robot falls below the lower threshold. If this occurs, it will overwrite output of the explore AFSM to navigate the robot towards areas of higher light intensity. If the energy level of the robot is above an upper threshold, this AFSM is disabled and the robot exhibits the exploration behavior.

Since we use the light sensor both to sense for obstacle and the light level, we implement a time-slicing scheme for light sensor between these two functions as illustrated in Figure 17. The robot's internal state is divided into two logical phases - a **SIGNAL** phase and a **IDLE** phase. During the **SIGNAL** phase, the robot's IR transmitter will broadcast IR packets and the light sensor reads the reflection of the packet to determine if there is an obstacle in front of it i.e., proximity sensing. During the **IDLE** phase, the IR transmitter does not send out any signal and the light sensor reads the ambient light level.

3.6 Evolving The Robot Controller Using Genetic Algorithms

In this section, we describe the final piece of our implementation - genetic algorithm on the robot controller (without light seeking behavior) in an attempt to evolve the obstacle avoidance behavior. We make use of 8 state FSAs to represent the controller of the robot. The actions **Forward**, **Left**, **Right** and **Reverse** are

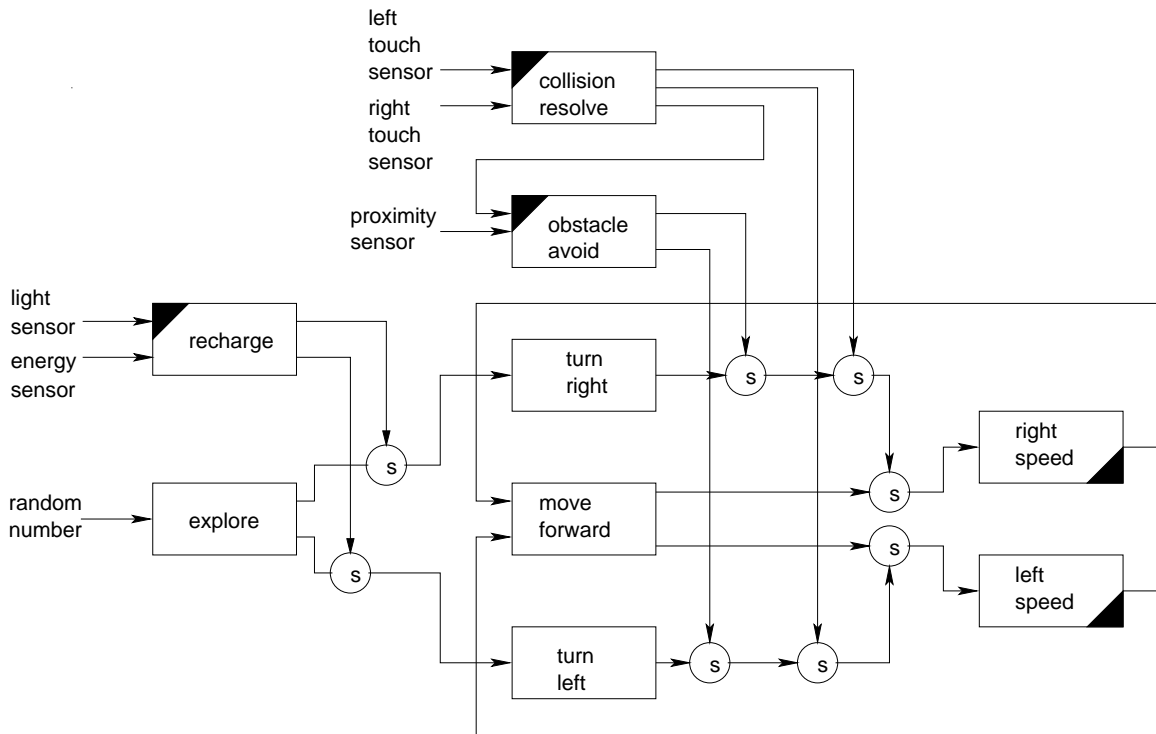


Figure 16: Subsumption Network Design with Light Seeking Behavior

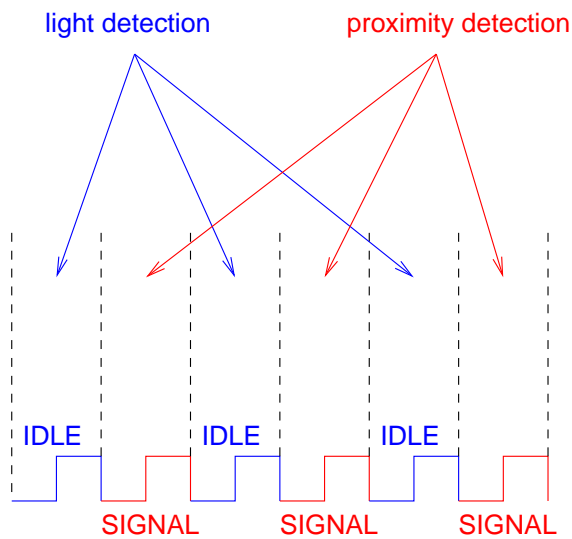


Figure 17: Time Slicing of Light Sensor Between Proximity and Light Sensing

| Parameter | Value |
|------------------------------------|--------------|
| Population Size | 5 |
| Generations | 10 |
| Probability of Choosing Individual | 50% |
| Steps/Iteration | 100 |
| Probability of Mutation | 1% |
| Proportion of Population Mutated | 40% |
| Crossover Points | 1 |
| Proportion of Population Crossover | 40% |

Table 5: Parameters for Genetic Algorithm

encoded (in binary) as 00, 01, 10 and 11 respectively. The robot's genotype (controller) is obtained by first ordering the $\langle \text{Old_State}, \text{Input} \rangle$ pairs lexicographically. The robot's genotype is obtained by concatenating the corresponding $\langle \text{Next_State}, \text{Action} \rangle$. The Input value is 1 if the robot's proximity sensor senses an obstacle, 0 otherwise.

Table 5 shows the list of parameters used in the genetic algorithm that we implemented on the robot. The robot is simulated for 100 exploration time steps for each controller i.e., the Explore AFSM is executed 100 times. At each simulation, the robot is initialized with an initial fitness of 100. If it hits an obstacle, the robot's fitness is decreased by 1. If the robot moves forward, its fitness is increased by 1. Otherwise, its fitness value remains the same. The robot stops after each simulation and will start the next simulation upon user input (via the RCX's PRGM button i.e., we overwrote the function of the PRGM button).

4 Experiment & Results

4.1 Exploration Robot with Obstacle Avoidance

In our first implementation of the exploration robot with obstacle avoidance as outline in Section 3.4, the robot learns how to avoid obstacles through trial-and-error i.e., by bumping into obstacles and adjusting its proximity threshold. As expected, the robot was able to learn how to avoid obstacles after bumping into obstacles a few times. One main limitation was that the robot has only one light sensor facing the forward direction but it has two bumpers on either side. This inevitably cause the robot to miss some obstacles with its proximity detection, causing the robot to run into obstacles if they appear in the light sensor’s blind spot.

We eventually provided the robot with two light sensors, one on the left side and one on the right side and this improved the robot’s obstacle avoidance behavior. However, what was most interesting is when the robot’s proximity detection function gets over sensitized due to the “accidental” bumping when it has only one light sensor. When the proximity sensor gets over-sensitized, the robot will wrongly detect an obstacle if the light intensity changes rapidly. This is due to the robot moving from an area with high light intensity to an area of low light intensity or vice versa. This will cause the robot to wrongly detect an obstacle, and preventing it from moving into another area of different lighting condition. We describe this type of behavior as the *comfort zone* syndrome. This syndrome can also be observed in the human eye when the light conditions that the eye perceives changes suddenly. A certain amount of discomfort and momentarily blindness will be observed by the eye. This emergent behavior is however, an artifact of the way we implemented our proximity sensor. If, for example, a sonar sensor is used instead, such behavior will not occur.

Another observation that we made during the experiments is that this robot exhibit deadlock conditions in certain situations. For example, when the robot moves into a corner, its right bumper may collide with the wall and the Collision-Resolve AFSM will cause the robot to turn left. When this happens, the robot’s left sensor will collide with the other wall, causing it to turn right again. This goes on for a prolonged period of time until the physical inaccuracies of the control breaks this deadlock e.g., the robot may travel longer distances when turning in one direction than the other although the power supplied to the actuators and the time allowed for turning is exactly the same. The problem become more acute when the robot enters into the recharge mode because the Recharge AFSM may keep making the robot turn left after a collision on the left bumper turns the robot right. In the exploration mode, the randomness of movement also helps to break this deadlock fairly quickly. We modified our robot controller to record the last action that the robot was doing when it collided with an obstacle. When it collides with an obstacle while moving forward, then the robot will reverse and turn away (in the opposite direction as the obstacle). When it collides with an obstacle while it is turning, the robot will reverse and continue its turning action, regardless of which bumper collided with the obstacle. This modification helped to eliminate the deadlock condition. This behavior is illustrated

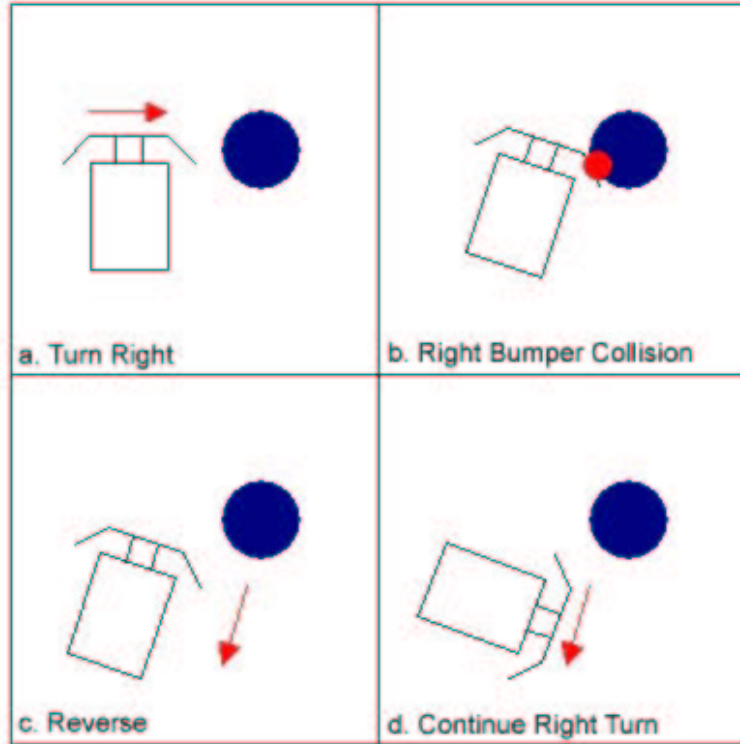


Figure 18: Robot Resolving A Right Collision While Turning Right

in Figure 18.

4.2 Light Seeking Controller

Our next implementation adds a light seeking behavior and a two-mode operation outlined in Section 3.5. This implementation uses a tree-like data structure that is similar in concept to eligibility traces used in Q-learning algorithms. This data structure is not without disadvantages. In our experiment, we found that the robot's movement in this algorithm is inherently biased towards turning in one direction. In our implementation, the robot's movement is biased towards turning to its left. However, the only problematic case is when the light source is slightly towards the right of the robot (less than 45° from its forward direction such that the highest light intensity observed by the robot's sensor is still directly in front). We use states 14, 15 and 16 (the new data structure is shown in Figure 19 to remedy the problems caused by this special case - when the maximum light intensity is in front of the robot, the robot will turn right one step to see if the direction towards its right offers greater light levels. These states are redundant when the left and right turns are at the initial values of 90° but comes into play when the robot is zeroing into the light source i.e.,

angle of turning is less than 90° . This redundancy is a tradeoff of having to keep two almost identical tree structure, one for the initial move and one for the subsequent moves.

Our experiments also showed that the robot did not react to changing conditions quickly. When the light that the robot navigate towards disappears, it takes a long time for the robot to make amendments to its path because the robot's angle of turning is reduced substantially due to the fine-tuning of direction. Hence the closer the robot is to the light source, the longer it takes for it to "recover" when the light source disappears. This is interestingly very similar to human behaviors [8]. To solve this problem, we provided the robot with the ability to adapt to sudden changes in light conditions e.g., the light source suddenly disappears. The robot keeps a three bit history vector that indicates if the robot's last two moves results in progressively lower light levels detected by its light sensor. Every time this is true, the robot would swing its angle of search to 90° . We could have increased this angle of turn by a smaller factor e.g., multiply by two but this does not help because we are still partially relying on outdated experiences of the robot. Therefore, we made a design decision to swing the robot's angle of search back to 90° , effectively asking the robot to restart its search for the light source.

Apart from the standalone behavior of the light seeking behavior, the interaction between the different AFSM in the subsumption architecture also provided us with its fair share of problems. The most significant problem arise when the robot collides with an obstacle (or avoids an obstacle). The robot does this by moving backward and turning away from the obstacle. The effect of this is that the robot gets thrown out of its original course in the direction of highest light level. As we do not know how long the robot would have turned before the collision, we cannot determine the actual angle that the robot must turn after the collision in order to compensate for the turning caused by the collision. We did not want to reset the robot's turning angle to 90° because this would take too long and may not be an acceptable solution especially if the robot is running out of energy. Incidentally, the solution to the deadlock problem described in Section 4.1 provided us with a good way of approximately compensating for the collision. For example, when the robot turns collides on turning left, it will move back and then continue to turn left. We argue that this continuation of its last movement actually provides an approximation to the turning action that it was executing before the collision. Experiments showed that our argument is valid unless there are simply too many obstacles in the way that the robot keeps bumping into things on every few moves.

4.3 Evolving the Robot's Controller

In building the robot, we realize that fine tuning the robot's parameters such as threshold values takes a lot of time and effort and gets increasingly difficult when the robot's controller becomes more complex. We further experimented with genetic algorithms in an attempt to evolve a robot controller to avoid obstacles.

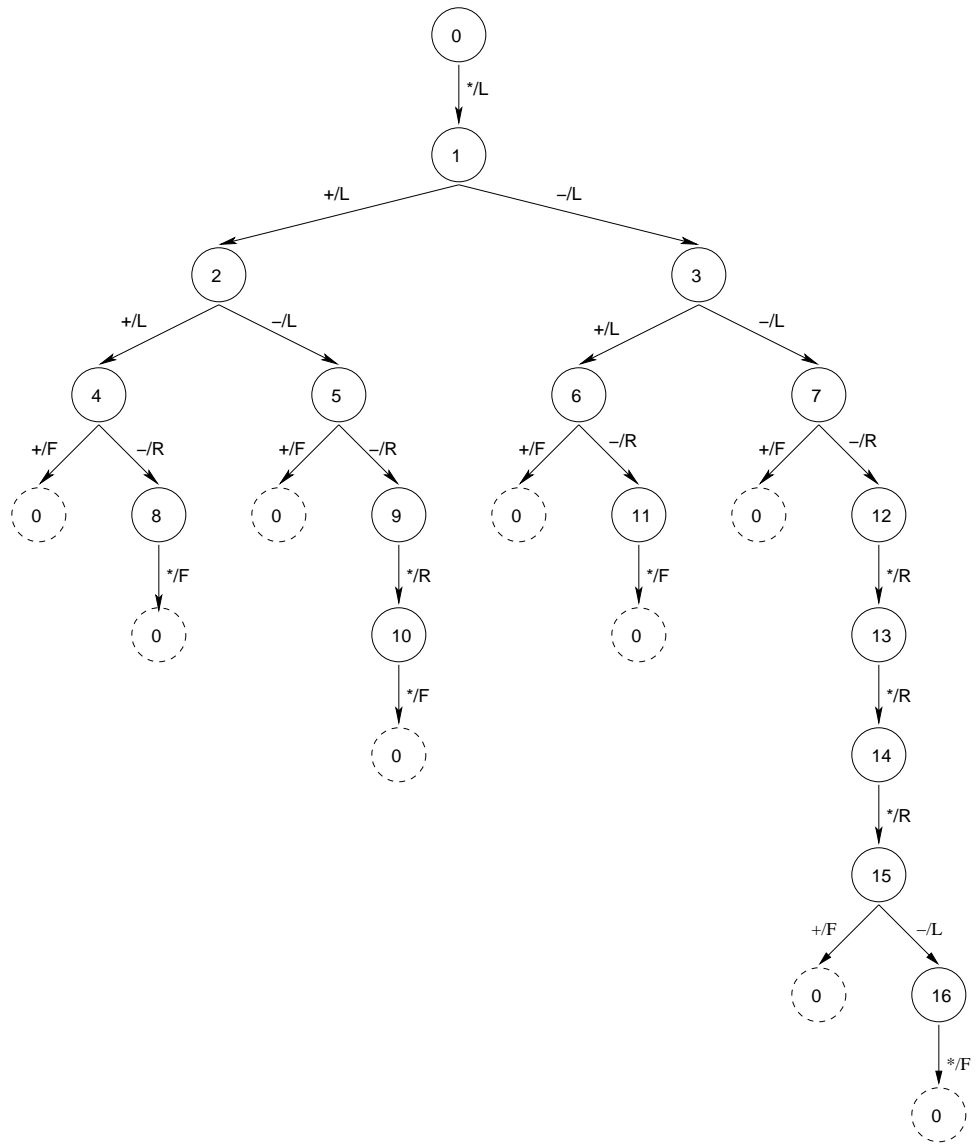


Figure 19: Tree-Like Structure Implementing Light Seeking Behavior Controller

This experiment was not complete but serves as a baseline for using genetic algorithms on our robot implementation.

We implemented a genetic algorithm on top of the the simple version of the robot controller. The input to the controller is a binary value that tells if there is an obstacle in front of the robot. We overwrite the RCX's PRGM button such that the robot will stop after each simulation and this button will have to be pushed for the robot to start the next simulation (of the next individual). The robot is initialized with a fitness of 100 and this value is decreased when the robot collides with an obstacle. The robot's fitness is incremented only when it moves forward; otherwise, the robot could keep turning in circles and still achieve optimal fitness.

5 Conclusion & Future Work

In this project, we implemented a basic robot controller that will adapt to its surroundings and learns how to avoid obstacles given its own internal condition (e.g., battery power) and the external environment that it operates in. We furthered the project by adding a light seeking behavior to the robot and allow the robot to navigate towards regions of high light intensity when its internal energy level is low. We implemented this behavior of learning to navigate towards the direction of highest light intensity using a tree-like data structure for representing the state-action pairs. This method is very similar in concept to eligibility traces. Eligibility traces are often applied on Q-learning reinforcement learning algorithms to overcome the problem that most real-life environment is not Markov. Applications for this behavior include recharging the robot using some form of solar energy panel and keeping the robot within communication range with a base station when both bodies are constantly moving (e.g., when the signal from the base station is low, the robot will navigate towards regions of stronger signals; otherwise, the robot will explore its environment). The robot was implemented using a subsumption architecture that allows its various behaviors such as obstacle avoidance, collision resolution, light seeking and exploring to interact with each other.

Our experiments reinforced our knowledge that building a physical robot that operates in the real world environment is very different from simulation e.g., of Braitenberg vehicles. There are a variety of reasons for this phenomenon, including both external and internal (to the robot) factors. External factors include:

- **Friction:** When the robot is run on different floorings, its behavior can vary e.g., its actions become sluggish on carpeted floor but becomes more agile on a hardwood floor. Friction also affects the speed and angular velocity of the moving robot.
- **Lighting conditions:** This affects the light sensor that was used in proximity detection. Sudden change in light conditions, for example, can cause the robot to inaccurately detect an obstacle where there is actually none in front of it.
- **Varying shapes and sizes of obstacles:** We have implemented obstacle avoidance and collision resolution behaviors in the robot. These however, can run into some peculiar cases where the obstacle cannot be easily detected by the proximity sensor or the collision detectors. There are also cases where the robot will be caught in a deadlock condition.

There are also some internal factors that affects the robot's operation. These include:

- **Sensor inaccuracies:** Commercial sensors that we used on the robot is neither precise nor accurate. These sensors often can give variable readings even when the robot remains motionless in a constant environment. It is hence important for us find out the level of accuracy and precision that the sensors (and the I/O ports) operate with, and also test the actual operation of the sensors in order to calibrate

the robot's movement. It is also important to note that that sensors do not present a complete picture of the external world and there are costs (e.g., time-slice) associated with using these sensors.

- **Battery power:** One constant problem that we encountered when building the robot is that the speed and angular velocity of the robot decreases with the remaining power in the batteries used in the RCX. This caused us to have to constantly recalibrate the time required for turning the robot left and right in order to achieve a certain angular displacement that we require.
- **Concurrent tasks:** Most robot implementation have some form of concurrent tasks to control the various behaviors of the robots. Too many concurrent tasks proved complex and difficult to get right. Furthermore, since we have only a single processor that implements a time-slicing scheme among the different tasks and the robot is required to react quickly at times, timing became a major issue i.e., insufficient time window to be shared among all the different tasks.
- **Software abstraction:** Software abstraction makes programming a much easier task. The problem with software abstraction is that it tends to abstract away some important timing and synchronization issues in programming the robot. For example, the NQC that we used is less powerful as compared to the legOS environment, which allows you to program in the standard ANSI C/C++ and utilize all 32 KB of RAM rather than limiting us to the number of variables as defined by the microcontroller. However, we found that NQC's task control is less erratic than that of legOS. There are also a couple of pitfalls in using legOS. Firstly, legOS has a priority inversion problem [11] that we have to solve by implementing our own synchronization primitives. Secondly, legOS's floating point calculation takes approximately twice as integer operations. We took care not to use floating point numbers in our legOS program.

In summary, building a real robot requires a lot of experimentation and fine tuning through trial-and-error. Furthermore, a change in the physical environment often required us to recalibrate the robot. Such fine tuning work is often a tedious and time-consuming task, and is very difficult to get it right. This is essentially why we implemented some form of online adaptation and learning technique to allow the robot to adjust to its physical environment while achieving its goals. We also experimented with using genetic algorithms to evolve the robot controller to avoid obstacle. Genetic algorithms has the advantage that deliberate programming and control fine-tuning on the part of the programmer is not required.

Even though genetic algorithm is an elegant way of "hands-off" learning that can be applied to a population of individuals over many generations, it is nevertheless, a time consuming process to simulate and evolve the robot controller. As we expected, applying genetic algorithm did not provide us with satisfactory results, primarily because we are limited to a small number of individuals over a small generation for our implementation to be feasible. On the average, simulating a single individual took about 5 to 8 minutes in our

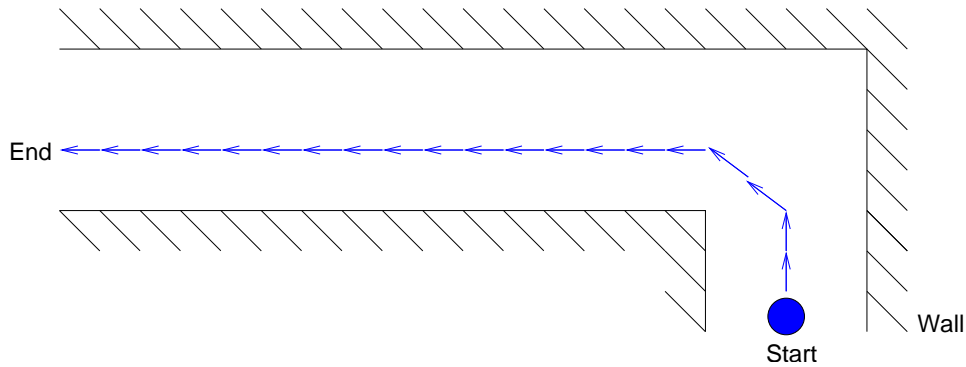


Figure 20: Best Evolved Robot Controller’s Approximate Behavior

implementation and the entire experiment took almost 10 hours to complete using our very small population size of 5 and over just 10 generations. Over the generations, the controller will have a tendency to move forward because the controller’s fitness is increased whenever it moves forward. The best controller did exhibit some form of memory of the floor plan of the environment that we conducted the simulation i.e., it turns away from the obstacle (the wall) even before reaching it and then starts to go straight to obtain maximum fitness. Figure 20 shows the best controller’s approximate behavior.

We believe that this behavior is a repercussion of the way we define our fitness function i.e., the robot is rewarded when it moves forward and is punished when it hits an obstacle. What begun as an attempt to evolve a controller that learns how to avoid obstacles ended up with a one that attempts to memorize the specific floor plan that it runs in using the robot’s internal states. In addition, since the robot is not rewarded for turning left or right, the robot does not exhibit too much exploration behavior. Finally, as we have seen in evolving “ants” [6], our simulation does not provide the controller with adequate population size, generations and variety in the environment for it to generalize its behavior to avoid obstacles.

Although we have implemented various pieces of a learning robot, we believe that there are still some interesting issues that needs to be addressed. As mentioned earlier, the angular velocity and speed of the robot is affected by its battery level. One piece of work that we believe is interesting is to enable the robot to adapt to its own battery power so that it can adjust the time required for turning left and right, and moving forward. Consequently, the robot adjust its movement accordingly so that the distance traveled at every step is more or less constant. Another area of interesting work is to study into how various ways of defining the fitness function for the genetic algorithm would affect the robot’s emergent behavior. For example, the current setting is that the robot’s fitness is incremented by 1 when it moves forward and decremented by 1 when it collides into an obstacle. We could increment the robot’s fitness by $\frac{1}{2}$ when it turns left or right

to encourage exploration but not condone the robot to keep turning in circles. Finally, with our experience gained in building a single robot, we believe that the next step would be to build more robots - determining how multiple robot with simple behaviors and learning capabilities would interact is also an area of future work.

References

- [1] Bonabeau, E., Theraulaz, G., Deneubourg, J.-L., Aron, S., and Camazine, S. Self-Organization in Social Insects. *Trends in Ecology & Evolution*, 12:188-193, 1997.
- [2] Brooks, A. B. A Robot that Walks; Emergent Behavior for a Carefully Evolved Network. In Proceedings of *IEEE International Conference on Robotics and Automation*, Scottsdale, AZ, May 1989.
- [3] Brooks, A. B. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, Vol. 2, 1: 14-23, March 1986.
- [4] Carbonell, J. G., Knoblock, C. A., and Milton, S. PRODIGY: An Integrated Architecture for Planning and Learning. *Technical Report*, CMU-CS-89-189, October 1989.
- [5] Capaldi, E. A., Smith, A.D., Osborne, J.L., Fahrbach, S.E., Farris, S.M., Reynolds, D.R., Edwards, A.S., Martin, A., Robinson, G.E., Poppy, G.M., Riley, J.R. *Ontogeny of orientation flight in the honeybee revealed by harmonic radar*. *Nature* 403: 537-540, 2000.
- [6] Chuang-Hue Moh, Research Assignment 4, *6.836 Embodied Intelligence, Massachusetts Institute of Technology*, April 2002.
- [7] Collaborative Mobile Robots for High-Risk Urban Missions. Stanford University. <http://underdog.stanford.edu/tmr/>.
- [8] Johnson, S., Blanchard, K. Who Moved My Cheese: An A-Mazing Way to Deal with Change in Your Work and Your Life. Putnam Publishing Group, September 1998.
- [9] LegOS. <http://legos.sourceforge.net/>.
- [10] LeJOS. <http://lejos.sourceforge.net/>.
- [11] Pedersen, M. H., Klitgaard, M. and Thomas, C. Solving the Priority Inversion Problem in legOS. University of Aalborg, UK, May 2000.
- [12] Newell, A., and Simon, H. A. Computer science as empirical enquiry: Symbols and search. *Communications of the ACM*, Vol. 19, 3: 113-126, March 1976.
- [13] Not-Quite-C. <http://www.enteract.com/dbaum/nqc/>.
- [14] Peng, J. and Williams, R. J. Technical Note: Incremental Q-learning. *Machine Learning*, 22:283-290, 1996.
- [15] Smart, D. W., and Kaelbling, L., P. Making Reinforcement Learning Work on Real-Robots. Research Abstract, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2000.

- [16] Smart, D. W., and Kaelbling, L., P. Practical Reinforcement Learning in Continuous Spaces. In *Proceedings of the Sixteenth International Conference on Machine Learning*, 2000.
- [17] The Handy Board. <http://www.handyboard.com>.
- [18] The MIT Programmable Brick Project. <http://el.www.media.mit.edu/groups/el/projects/programmable-brick/>.
- [19] Watkins, C. J. C. H. Learning from Delayed Rewards. Ph.D. Thesis, King's College, Cambridge, England, 1989.
- [20] Wyatt J. Issues in putting Reinforcement Learning onto robots. *Mobile Robotics Workshop, 10th Biennial Conference of the AISB*, Sheffield, UK, 1995.

A NQC Codes for Learning Robot

```
/* Copyright 2002, Chuang-Hue Moh, Massachusetts Institute of Technology */

/* The lower the number, the higher the priority */
#define MIN_PRIORITY 255

/* For movement */
#define FORWARD 1
#define REVERSE -1
#define LEFT 2
#define RIGHT 3

/* For proximity */
#define DISTANCE_STEPS 10
#define DISTANCE_MAX 250
#define DISTANCE_MIN 50

/* IR transmitter signalling states */
#define IDLE 0
#define SIGNAL 1

/* Explore/recharge phases and paramters */
#define RECHARGE 1
#define EXPLORE 0
#define RECHARGE_THRESHOLD 700
#define MAX_RECHARGE 10
#define ENERGY_THRESH_LOW 100
#define ENERGY_THRESH_HIGH 200
#define MAX_ENERGY 400

/* Time for controlling angle of turn */
#define DEFAULT_TURN_TIME 190
#define DEFAULT_TIME 64

/* Task priority levels */
#define PRIO_EXPLORE MIN_PRIORITY
#define PRIO_RECHARGE MIN_PRIORITY-1
#define PRIO_ENERGY MIN_PRIORITY-2
#define PRIO_DETECT MIN_PRIORITY-3
#define PRIO_SIGNAL MIN_PRIORITY-4
#define PRIO_AVOID MIN_PRIORITY-5
#define PRIO_LIGHT MIN_PRIORITY-6

/* Variable for obstacle avoidance, tuned when collision occurs */
int proximity_threshold;

/* Turn time for light seeking, multiplicative decrease, additive increase */
int turn_time;
int last_action;

/* FSM state for light seeking */
int state;

/* Current mode = RECHARGE/EXPLORE */
int mode;

/* Robot's current energy level */
int energy;

/* Signalling phase = SIGNAL/IDLE */
int phase;

/* For remembering the light readings for the last 2 steps */
int read_history[3];
int read_turn;

/* For remembering the highest light intensity (lowest reading value) */
int min_reading;

task main() {
    init();
    start explore;
    start recharge;
    start energy_monitor;
    start collision_detect;
    start avoid_obstacle;
    start light_inidicator;
}

/*
Task for controlling the light indicator (energy level of the robot).
*/
task light_inidicator() {
    int last_energy;
    int display;
```

```

SetPriority(PRIO_LIGHT);
while(1) {
    last_energy = energy;
    SetUserDisplay(energy, 0);
    Wait(10);
    if ( energy > 0 ) {
        display = 7;
        if ( energy < 40 ) {
            display = 0;
        }
        else if ( energy >= 40 && energy < 80 ) {
            display = 1;
        }
        else if ( energy >= 80 && energy < 120 ) {
            display = 2;
        }
        else if ( energy >= 120 && energy < 160 ) {
            display = 3;
        }
        else if ( energy >= 160 && energy < 200 ) {
            display = 4;
        }
        else if ( energy >= 200 && energy < 240 ) {
            display = 5;
        }
        else if ( energy >= 240 && energy < 280 ) {
            display = 6;
        }
        SetPower(OUT_B, display);
        OnFwd(OUT_B);
    }
    else {
        Off(OUT_B);
    }
    until( last_energy != energy );
}
}

/*
Task to monitor the energy level of the robot.
Sets mode to EXPLORE if energy > high threshold
and to RECHARGE if energy < low threshold where
high threshold > low threshold.
*/
task energy_monitor() {
    SetPriority(PRIO_ENERGY);
    while ( 1 ) {
        until ( energy < ENERGY_THRESH_LOW );
        mode = RECHARGE;
        until ( energy > ENERGY_THRESH_HIGH );
        mode = EXPLORE;
        turn_time = DEFAULT_TURN_TIME;
        state = 0;
        read_turn = 0;
        read_history[0] = 999;
        read_history[1] = 999;
        read_history[2] = 999;
    }
}

/*
Task for sending IR signals via the IR transmitter
*/
task send_signal() {
    SetPriority(PRIO_SIGNAL);
    while ( 1 ) {
        phase = IDLE;
        Wait(10);
        phase = SIGNAL;
        SendMessage(0);
        Wait(10);
    }
}

/*
Task to avoid obstacle by reading the values of the
reflected IR packets and determine if the delta between
two consecutive readings is large enough to constitute
for an obstacle detection.
*/
task avoid_obstacle() {
    int previous_proximity, diff;
    SetPriority(PRIO_AVOID);
    start_send_signal;
    while ( 1 ) {
        until ( phase == SIGNAL );
        previous_proximity = SENSOR_2;
    }
}

```

```

diff = SENSOR_2 - previous_proximity;
if ( diff < 0 )
    diff = -diff;
if ( diff > proximity_threshold ) {
    acquire(ACQUIRE_OUT_A + ACQUIRE_OUT_C) {
        move_backward(DEFAULT_TIME);
        if ( last_action == LEFT )
            turn_left(DEFAULT_TIME);
        else
            turn_right(DEFAULT_TIME);
    }
    last_action = NONE;
    PlayTone(262, 10);
    PlayTone(294, 10);
    PlayTone(330, 10);
}
}
}

/*
Task to detect if a collision has occurred by reading the touch
sensor values.
*/
task collision_detect() {
    int rand;
    SetPriority (PRIO_DETECT);
    while ( 1 ) {
        until ( SENSOR_1 == 1 || SENSOR_3 == 1 );
        acquire (ACQUIRE_OUT_A + ACQUIRE_OUT_C) {
            collision_resolve();
        }
        proximity_threshold -= DISTANCE_STEPS;
        if ( proximity_threshold < DISTANCE_MIN )
            proximity_threshold = DISTANCE_MIN;
    }
}

/*
Task for light seeking behavior. Only kicks in if the current
mode is RECHARGE. Decreases the turn time by a factor of two
at each step. Automatically recharges if the light intensity
is greater than the RECHARGE THRESHOLD.
*/
task recharge() {
    int rand, cur_reading, input, action, recharge_amt;
    SetPriority (PRIO_RECHARGE);
    while ( true ) {
        until ( mode == RECHARGE );
        until ( phase == IDLE );
        cur_reading = SENSOR_2;
        if ( cur_reading < RECHARGE_THRESHOLD ) {
            recharge_amt = RECHARGE_THRESHOLD - cur_reading;
            recharge_amt = ( recharge_amt > MAX_RECHARGE )
                ? MAX_RECHARGE : recharge_amt;
            energy += recharge_amt;
            energy = ( energy > MAX_ENERGY ) ? MAX_ENERGY : energy;
        }
        if ( state == 0 ) {
            read_history[read_turn] = cur_reading;
            read_turn = (read_turn+1)%3;
            if ( read_turn == 0 ) {
                if ( (read_history[0] < read_history[1]) &&
                    (read_history[1] < read_history[2]) ) {
                    turn_time = DEFAULT_TURN_TIME;
                }
            }
        }
        if ( cur_reading > min_reading ) {
            input = 0;
        }
        else {
            input = 1;
            min_reading = cur_reading;
        }
        get_action(state, input, action);
        get_next_state(state, input);
        last_action = NONE;
        acquire (ACQUIRE_OUT_A + ACQUIRE_OUT_C) {
            switch(action) {
                case LEFT:
                    turn_left(turn_time);
                    break;
                case RIGHT:
                    turn_right(turn_time);
                    break;
                case FORWARD:
                    min_reading = 999;
            }
        }
    }
}

```

```

        turn_time = (turn_time >= 100) ? turn_time/2 : 50;
        move_forward(DEFAULT_TIME);
        break;
    }
}
catch {
    last_action = action;
}
}
}
}
/*
Task for exploring the environment. Uses a random number for
determining the next move. Subsumed by recharge task when the
mode is not EXPLORE. Automatically recharges if the light intensity
is greater than the RECHARGE THRESHOLD.
*/
task explore() {
    SetPriority (MIN_PRIORITY);
    int rand, cur_reading, recharge_amt;
    while ( true ) {
        until( mode == EXPLORE );
        if ( phase == IDLE ) {
            cur_reading = SENSOR_2;
            if ( cur_reading < RECHARGE_THRESHOLD ) {
                recharge_amt = RECHARGE_THRESHOLD - cur_reading;
                recharge_amt = ( recharge_amt > MAX_RECHARGE )
                    ? MAX_RECHARGE : recharge_amt;
                energy += recharge_amt;
                energy = ( energy > MAX_ENERGY ) ? MAX_ENERGY : energy;
            }
        }
        last_action = NONE;
        acquire (ACQUIRE_OUT_A + ACQUIRE_OUT_C) {
            rand = Random(10);
            switch( rand ) {
                case 1: case 2:
                    turn_right(2*DEFAULT_TIME);
                    break;
                case 3: case 4:
                    turn_left(2*DEFAULT_TIME);
                    break;
                default:
                    move_forward(2*DEFAULT_TIME);
                    break;
            }
        }
        catch {
            switch( rand ) {
                case 1: case 2:
                    last_action = RIGHT;
                    break;
                case 3: case 4:
                    last_action = LEFT;
                    break;
                default:
                    last_action = FORWARD;
                    break;
            }
        }
    }
}
}
/*
Function to resolve collision. Increases the turn time when a
collision occurs.
*/
void collision_resolve() {
    int s1 = SENSOR_1, s3 = SENSOR_3;
    if ( last_action == LEFT ) {
        turn_left(DEFAULT_TIME);
    }
    else if ( last_action == RIGHT ) {
        turn_right(DEFAULT_TIME);
    }
    else if ( s1 == 1 && s3 == 0 ) {
        move_backward(DEFAULT_TIME);
        turn_right(DEFAULT_TIME);
        move_forward(DEFAULT_TIME);
    }
    else if ( s3 == 1 ) {
        move_backward(DEFAULT_TIME);
        turn_left(DEFAULT_TIME);
        move_forward(DEFAULT_TIME);
    }
    last_action = NONE;
}
}

```

```

/*
For light seeking action (implement eligibility trace).
Given the <state, input> pair, the function returns the action.
*/
void get_action(int state, int input, int &action) {
    if( input == 0 ) {
        switch( state ) {
            case 0: case 1: case 2: case 3: case 15:
                action = LEFT;
                break;
            case 4: case 5: case 6: case 7:
                action = RIGHT;
                break;
            case 8: case 10: case 11: case 16:
                action = FORWARD;
                break;
            case 9: case 12: case 13: case 14:
                action = RIGHT;
                break;
        }
    }
    else if ( input == 1 ) {
        switch ( state ) {
            case 0: case 1: case 2: case 3:
                action = LEFT;
                break;
            case 4: case 5: case 6: case 7:
                action = FORWARD;
                break;
            case 8: case 10: case 11: case 15: case 16:
                action = FORWARD;
                break;
            case 9: case 12: case 13: case 14:
                action = RIGHT;
                break;
        }
    }
}

/*
For light seeking action (implement eligibility trace).
Given the <state, input> pair, the function returns the next state.
*/
void get_next_state(int &state, int input) {
    switch( state ) {
        case 0:
            state = 1;
            break;
        case 1: case 2: case 3:
            state = ( input == 1 ? state*2 : state*2+1 );
            break;
        case 4:
            state = ( input == 1 ? 0 : 8 );
            break;
        case 5:
            state = ( input == 1 ? 0 : 9 );
            break;
        case 6:
            state = ( input == 1 ? 0 : 11 );
            break;
        case 7:
            state = ( input == 1 ? 0 : 12 );
            break;
        case 8: case 10: case 11: case 14:
            state = 0;
            break;
        case 9: case 12: case 13: case 16:
            state = state+1;
            break;
        case 15:
            state = ( input == 1 ? 0 : 16 );
            break;
    }
}

/* -----
Initialization function.
Note:
Action      Desc
0           Left
1           Right
2           Forward
3           Left (time = time/2)
4           Right (time = time/2)
5           Forward (time = time/2)
----- */

```

```

void init() {
    read_turn      = 0;
    state          = 0;
    phase          = IDLE;
    turn_time      = DEFAULT_TURN_TIME;
    energy         = 200;
    mode           = EXPLORE;
    min_reading    = 999;
    read_turn      = 0;
    read_history[0] = 999;
    read_history[1] = 999;
    read_history[2] = 999;
    proximity_threshold = DISTANCE_MAX;
    last_action    = NONE;
    SetSensor(SENSOR_1, SENSOR_TOUCH);
    SetSensor(SENSOR_3, SENSOR_TOUCH);
    SetSensor(SENSOR_2, SENSOR_LIGHT);
    SetSensorMode(SENSOR_2, SENSOR_MODE_RAW);
}

/* Move robot forward */
void move_forward(int twait) {
    left_motor_on(FORWARD);
    right_motor_on(FORWARD);
    Wait(twait);
    left_motor_off();
    right_motor_off();
}

/* Move robot backwards */
void move_backward(int twait) {
    left_motor_on(REVERSE);
    right_motor_on(REVERSE);
    Wait(twait);
    left_motor_off();
    right_motor_off();
}

/* Turn robot right */
void turn_right(int twait) {
    left_motor_on(FORWARD);
    right_motor_on(REVERSE);
    Wait(twait);
    left_motor_off();
    right_motor_off();
}

/* Turn robot left */
void turn_left(int twait) {
    left_motor_on(REVERSE);
    right_motor_on(FORWARD);
    Wait(twait);
    left_motor_off();
    right_motor_off();
}

/* Switch on right motor */
void right_motor_on(int fwd) {
    if ( energy > 0 ) {
        SetPower (OUT_A, OUT_FULLL);
        if ( fwd == FORWARD )
            OnFwd(OUT_A);
        else
            OnRev(OUT_A);
        energy--;
    }
}

/* Switch right motor off */
void right_motor_off() {
    Off(OUT_A);
}

/* Switch on left motor */
void left_motor_on(int fwd) {
    if ( energy > 0 ) {
        SetPower (OUT_C, OUT_FULLL);
        if ( fwd == FORWARD )
            OnFwd(OUT_C);
        else
            OnRev(OUT_C);
        energy--;
    }
}

/* Switch left motor off */

```

```
void left_motor_off() {  
    Off(OUT_C);  
}
```

B LegOS Codes for Evolving Robot Control

```
/* Copyright 2002, Chuang-Hue Moh, Massachusetts Institute of Technology */
```

```
#include <conio.h>
#include <unistd.h>
#include <dsensor.h>
#include <dmotor.h>
#include <dbutton.h>
#include <sys/time.h>
#include <stdlib.h>
#include <sys/tm.h>

/* robot control */
#define FWD_SPEED MAX_SPEED
#define REV_SPEED MAX_SPEED
#define PRIO_AVOID PRIO_LOWEST+2
#define PRIO_EXPLORE PRIO_LOWEST+2
#define PRIO_RESOLVE PRIO_LOWEST+3
#define PROX_THRESH 0x30
#define IDLE 0
#define SIGNAL 1

pid_t pid0, pid1, pid2, pid3, pid4, pid5;
int ready = 1;
unsigned proximity = 0, cur_pri = PRIO_LOWEST, phase = IDLE, detecting = 0;

/* GA parameters */
#define state_bits 3
#define states 8
#define action_bits 2
#define popsize 5
#define generations 10
#define iterations 100
#define choose 50
#define mutate 1
#define crossover_num (int)(0.4*popsize)
#define mutate_num (int)(0.4*popsize)
#define ctrl_bits (states<<1)*(state_bits+action_bits)

unsigned state = 0;
int fitness = 0;

typedef struct controller_t {
    int fsa[ctrl_bits];
    int fitness;
} controller;

int cur_rank[popsize];
int new_rank[popsize + crossover_num + mutate_num];
controller* cur_pop[popsize];
controller* new_pop[popsize + crossover_num + mutate_num];
int new_popsize;
controller *cur_ctrl;

void decode(controller *, int *, int *, int, int);
void init_arrays();
void free_arrays();
void rank_ctrl();
void rank_ctrl_new();
int select_ctrl();
void crossover_ctrl();
void mutate_ctrl();
void select_next_generation();
void init_new_generation();
void move_forward();
void turn_left();
void turn_right();
void reverse();
void stop(int sleep);
wakeupt_t turn(wakeupt_t);
wakeupt_t finish(wakeupt_t);
wakeupt_t collision(wakeupt_t);
int resolve();
int explore();
int ticker();
int signal();
int avoid();
void simulate(controller *);
void set_control(controller *);
controller* get_control();
void motor_control(int, int, int, int, int);

/*
Function for decoding the <next_state, action> pair given the
<current_state, input> tuple.
Note encoding:
*/
```

```

        00 - forward    01 - left
        10 - right     11 - nop
*/
void decode( controller *ctrl, int *action, int *next_state,
            int input, int state ) {
    int i;
    int address = ((state<<1)+input)*(state_bits+action_bits);
    *action = (ctrl->fsa[address+state_bits]<<1)
        + ctrl->fsa[address+state_bits+1];
    *next_state = 0;
    for ( i = 0; i < state_bits; i++ )
        *next_state += ctrl->fsa[address+i] << (state_bits-i-1);
}

/*
Function for initializing the array used in the genetic algorithms
*/
void init_arrays() {
    int i, j;
    for ( i = 0; i < popsize; i++ ) {
        cur_pop[i] = (controller*)malloc(sizeof(controller));
        for ( j = 0; j < ctrl_bits; j++ ) {
            cur_pop[i]->fsa[j] = random()%2;
        }
    }
    for ( i = 0; i < popsize + crossover_num + mutate_num; i++ ) {
        if ( i < popsize )
            new_pop[i] = cur_pop[i];
        else
            new_pop[i] = NULL;
    }
}

/*
Function for freeing the memory used by the array
*/
void free_arrays() {
    int i;
    for ( i = 0; i < popsize; i++ ) {
        if ( cur_pop[i] ) {
            free( cur_pop[i] );
            cur_pop[i] = NULL;
        }
    }
}

/*
Function that ranks the controller based on its fitness
(for current generation).
*/
void rank_ctrl() {
    int i, j, index;
    int max_fit;
    int tmp_fit[popsize];

    for ( i = 0; i < popsize; i++ ) {
        tmp_fit[i] = cur_pop[i]->fitness;
    }

    for ( i = 0; i < popsize; i++ ) {
        max_fit = -1;
        index = -1;
        for ( j = 0; j < popsize; j++ ) {
            if ( tmp_fit[j] > max_fit ) {
                max_fit = tmp_fit[j];
                index = j;
            }
        }
        cur_rank[i] = index;
        tmp_fit[index] = -1;
    }
}

/*
Function that ranks the controller based on its fitness
(for new generation).
*/
void rank_ctrl_new() {
    int i, j, index;
    int max_fit;
    int total = popsize + crossover_num + mutate_num;
    int tmp_fit[total];

    for ( i = 0; i < total; i++ ) {
        tmp_fit[i] = new_pop[i]->fitness;
    }
}

```

```

for ( i = 0; i < total; i++ ) {
    max_fit = -1;
    index = -1;
    for ( j = 0; j < total; j++ ) {
        if ( tmp_fit[j] > max_fit ) {
            max_fit = tmp_fit[j];
            index = j;
        }
    }
    new_rank[i] = index;
    tmp_fit[index] = -1;
}
}

/*
Function for selecting the controller based on the variable
``choose``.
*/
int select_ctrl() {
    int i;
    while ( 1 ) {
        for ( i = 0; i < popsize; i++ ) {
            if ( random()%100 <= choose )
                return cur_rank[i];
        }
    }
}

/*
Function that implements the crossover between two controllers
and inserts the new controller into the new generation.
*/
void crossover_ctrl() {
    int cp = random()%ctrl_bits;
    int i, a, b;
    controller *ctrl_a, *ctrl_b, *new_ctrl;
    a = select_ctrl();
    b = select_ctrl();
    ctrl_a = cur_pop[cur_rank[a]];
    ctrl_b = cur_pop[cur_rank[b]];
    new_ctrl = (controller *)malloc(sizeof(controller));
    new_pop[new_popsize] = new_ctrl;
    for ( i = 0; i < cp; i++ ) {
        new_ctrl->fsa[i] = ctrl_a->fsa[i];
    }
    for ( ; i < ctrl_bits; i++ ) {
        new_ctrl->fsa[i] = ctrl_b->fsa[i];
    }
    cputs("ready");
    ds_passive(&SENSOR_2);
    wait_event(&dkey_released, KEY_PRGM);
    msleep(250); /* debounce the button */
    wait_event(&dkey_pressed, KEY_PRGM);
    ds_active(&SENSOR_2);
    simulate(new_ctrl);
    new_popsize++;
}

/*
Function that implements the mutation of a controller
and inserts the new controller into the new generation.
*/
void mutate_ctrl() {
    int i, a;
    controller *ctrl_a, *new_ctrl;
    a = select_ctrl();
    ctrl_a = cur_pop[cur_rank[a]];
    new_ctrl = (controller *)malloc(sizeof(controller));
    new_pop[new_popsize] = new_ctrl;
    for ( i = 0; i < ctrl_bits; i++ ) {
        if ( random()%100 <= mutate )
            new_ctrl->fsa[i] = 1- ctrl_a->fsa[i];
        else
            new_ctrl->fsa[i] = ctrl_a->fsa[i];
    }
    cputs("ready");
    ds_passive(&SENSOR_2);
    wait_event(&dkey_released, KEY_PRGM);
    msleep(250); /* debounce the button */
    wait_event(&dkey_pressed, KEY_PRGM);
    ds_active(&SENSOR_2);
    simulate(new_ctrl);
    new_popsize++;
}
}
/*

```

```

    Function for selecting the next generation based on the
    controller's rank.
*/
void select_next_generation() {
    int i;
    sleep(2);
    for ( i = 0; i < popsize; i++ ) {
        cur_pop[i] = new_pop[new_rank[i]];
    }
    for ( ; i < new_popsize; i++ ) {
        free(new_pop[new_rank[i]]);
        new_pop[i] = NULL;
    }
}

/*
Function for initializing the new generation with the
current generation.
*/
void init_new_generation() {
    int i;
    int total = popsize + crossover_num + mutate_num;
    for ( i = 0; i < total; i++ ) {
        if ( i < popsize )
            new_pop[i] = cur_pop[i];
        else
            new_pop[i] = NULL;
    }
    new_popsize = popsize;
}

/* === Functions for robot control === */
void move_forward() {
    motor_control(fwd, fwd, FWD_SPEED, FWD_SPEED, 1000);
}

void turn_left() {
    motor_control(fwd, rev, FWD_SPEED, REV_SPEED, 1000);
}

void turn_right() {
    motor_control(rev, fwd, REV_SPEED, FWD_SPEED, 1000);
}

void reverse() {
    motor_control(rev, rev, FWD_SPEED, FWD_SPEED, 500);
    motor_control(rev, rev, FWD_SPEED/2, FWD_SPEED/2, 250);
    motor_control(rev, rev, MIN_SPEED, MIN_SPEED, 250);
}

void stop(int sleep) {
    motor_control(fwd, fwd, 0, 0, sleep);
}

void motor_control( int dir_a, int dir_c, int speed_a, int speed_c,
                    int sleep ) {
    motor_a_dir(dir_a);
    motor_c_dir(dir_c);
    motor_a_speed(speed_a);
    motor_c_speed(speed_c);
    msleep(sleep);
}

/* === End of robot control function === */

/* === Synchronization primitives === */
/* Overcomes priority inversion problem in legOS */
wakeupt_t turn(wakeupt_t data) {
    int proc_pri = (int) data;
    if ( proc_pri >= cur_pri ) {
        cur_pri = proc_pri;
        return 1;
    }
    else {
        return 0;
    }
}

wakeupt_t finish(wakeupt_t data) {
    int proc_pri = (int) data;
    if ( proc_pri >= cur_pri ) {
        cur_pri = PRIO_LOWEST;
        return 1;
    }
    else {
        return 0;
    }
}

```

```

    }
}
/* === End of synchronization primitives === */

/*
Function that detects collision.
*/
wakeup_t collision(wakeup_t data) {
    if ( TOUCH_1 )
        return 1;
    else if ( TOUCH_3 )
        return 3;
    else
        return 0;
}

/*
Function that detects obstacles */
/*
wakeup_t obstacle(wakeup_t data) {
    return ( proximity > 0 );
}

/*
Function for resolving collisions when they occur.
*/
int resolve() {
    int event;
    while ( 1 ) {
        event = wait_event(&collision, 0);
        wait_event(turn, PRIO_RESOLVE);
        get_control()->fitness--;
        reverse();
        if ( event == 1 ) {
            turn_right();
        }
        else {
            turn_left();
        }
        stop(100);
        wait_event(&finish, PRIO_RESOLVE);
    }
    return 1;
}

/* == Controller manipulation functions == */
void set_control(controller *ctrl) {
    cur_ctrl = ctrl;
}

controller* get_control() {
    return cur_ctrl;
}
/* == End of controller manipulation functions == */

/*
Function implementing the explore AFSM
*/
int explore() {
    int action, next_state;
    int i;
    ready = 0;
    for ( i = 0; i < iterations; i++ ) {
        wait_event(&turn, PRIO_EXPLORE);
        decode( get_control(), &action, &next_state,
            (proximity>0), state);
        proximity = 0;
        state = next_state;
        if ( action == 0 ) {
            get_control()->fitness++;
            move_forward();
        }
        else if ( action == 1 ) {
            turn_left();
        }
        else if ( action == 2 ) {
            turn_right();
        }
        else if ( action == 3 ) {
            stop(1000);
            reverse();
        }
        stop(10);
        wait_event(&finish, PRIO_EXPLORE);
    }
    ready = 1;
    return 1;
}

```

```

}

/*
Function for broadcasting IR packets.
*/
int signal() {
while(1){
S_SR&=~(SSR_TRANS_EMPTY | SSR_TRANS_END); // clean up Flags
S_CR|=SCR_TRANSMIT; // enable Transmitter
S_TDR = 0x00; // transmit as many lightpulses as possible
phase = SIGNAL; // Indicate transmitting
msleep(3); // wait a little while
phase = IDLE; // Indicate Idle
msleep(6); // and wait again
}
}

/*
Function for determining if the current phase is SIGNAL.
*/
wakeupt_phase_signal(wakeupt_data) {
return phase;
}

/*
Function for counting the reflected packets to determine if there is
an obstacle in front of the robot.
*/
int ticker() {
int cur=0, last=0, diff=0;
while(1){
proximity = 0;
while(phase == IDLE) {} /* wait for transmission */
while(phase == SIGNAL) {
detecting = 1;
last = cur;
cur = SENSOR_2>>7;
diff = last;
diff -= cur;
if(diff > PROX_THRESH) {
proximity++;
}
}
detecting=0;
}
}

/*
Function that implements the obstacle avoidance AFSM.
*/
int avoid() {
unsigned mode;
while( 1 ) {
wait_event(&obstacle, 0);
wait_event(&turn, PRIO_AVOID);
reverse();
mode = random()%2;
if ( mode == 0 ) {
turn_right();
}
else {
turn_left();
}
proximity=0;
stop(100);
wait_event(&finish, PRIO_AVOID);
}
return 1;
}

/*
Function for detecting the end of a single simulation.
*/
wakeupt_end_simulate(wakeupt_data) {
return ready;
}

/*
Function for simulating the robot controller.
*/
void simulate(controller *ctrl) {
ctrl->fitness = 100;
state = 0;
cur_pri = PRIO_LOWEST;
set_control(ctrl);
pid0 = execi(&explore, 0, NULL, PRIO_EXPLORE, DEFAULT_STACK_SIZE);
}

```

```

pid1 = execi(&resolve, 0, NULL, PRIO_RESOLVE, DEFAULT_STACK_SIZE);
/* pid2 = execi(&avoid, 0, NULL, PRIO_AVOID, DEFAULT_STACK_SIZE); */
pid3 = execi(&signal, 0, NULL, PRIO_AVOID, DEFAULT_STACK_SIZE);
pid4 = execi(&ticker, 0, NULL, PRIO_AVOID, DEFAULT_STACK_SIZE);
tm_start();
sleep(1);
wait_event(&end_simulate, 0);
stop(1000);
kill(pid0);
kill(pid1);
cur_pri = PRIO_LOWEST;
lcd_int(ctrl->fitness);
sleep(1);
}

int main() {
int gen, pop, i;
srandom(sys_time);
init_arrays();

S_BRR = B2400;
*((char*)&PORT4) &=~1;

/* Prolog */
for( pop = 0; pop < popsize; pop++ ) {
cputs("ready");
ds_passive(&SENSOR_2);
wait_event(&dkey_released, KEY_PRGM);
msleep(250); /* debounce the button */
wait_event(&dkey_pressed, KEY_PRGM);
ds_active(&SENSOR_2);
simulate(cur_pop[pop]);
}
rank_ctrl();
new_popsize = popsize;

for( gen = 0; gen < generations; gen++ ) {
init_new_generation();
for ( i = 0; i < crossover_num; i++ )
crossover_ctrl();
for ( i = 0; i < mutate_num; i++ )
mutate_ctrl();
rank_ctrl_new();
select_next_generation();
}

free_arrays();
return 1;
}

```