Development of control algorithms for a self-learning mobile robot

Ionut Resceanu, Marius Niculescu
Faculty of Automation, Computers and Electronics
University of Craiova
Bd. Decebal, Nr.107, 200440, Craiova
ROMANIA

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. obstacles, light). In this project, there were analized the possibilities to implement a robot that learns how avoid obstacle using online self-adaptation. Initially it was sudied and implemented a robot that explores an unknown path, using touch sensors and an obstacol detector to find its way during the exploration. Finally it was implemented a genetic algorithm on the robot and experimented with using genetic algorithm as a form of robot learning. The robot was built using the Lego RCX.

Key-Words: Robots; Learning; Genetic Algorithms; Mobile robotics; Lego RCX; control algorithms

1 Introduction

The goal for real-time mobile robots is to travel while avoiding obstacles in a navigation environment. Autonomous navigation allows robots to plan their path without the need for human intervention. The pathplanning problem has been shown to be difficult, thus this problem is often solved using heuristic optimization methods such as genetic algorithms. An important part of the genetic algorithm solution is the structure of the genotype that represents paths in the navigation environment. The genotype must represent a valid path, but still be simple to process by the genetic algorithm in order to reduce computational requirements. Unfortunately, many contemporary genetic path-planning algorithms use complex structures that require a significant amount of processing, which can affect the real-time response of the robot.

Mobile robots are desirable for operations such as bomb disposal or hazardous material management, which would be potentially dangerous for humans. An important task for the robot is autonomous navigation, where the robot travels between a starting point and a target point without the need for human intervention. While basic information may be available to the robot about the navigation area boundaries, unknown obstacles may exist within the navigation area. This is called an uncertain environment: the robot must be able to maneuver around these obstacles in order to reach its target point. The world space refers to the physical space in which robots and obstacles exist - the free space is the subset of the world space that is not occupied by obstacles. A path between the starting and target points that avoids collisions with obstacles is said to be feasible - this is a path that lies within free space. Thus, robot navigation methods need to solve the path-planning problem, which is to generate a feasible path and optimize this path with respect to certain criteria.

Applying learning algorithms such as machine learning and reinforcement learning to real physical robots is an area of active research in embodied intelligence. [15, 16] Autonomous, online learning robots possess the ability to operate in complex, dynamic environments through training and instruction to improve the robots' connection between its perception and action. These learning techniques are often superior to conventional programming because it is often difficult to design a robot that caters to all the variables in the real physical world.

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

In this project, it was implemented an autonomous, learning robot that performs exploration of the environment and learns how to avoid obstacle effectively. In other words, it was integrated sensor learning with robot control and implemented it on a real robot. The robot's controller is implemented

with the subsumption architecture that allows the robot's various behaviors (exploration, obstacle avoidance, collision resolution, etc.) to interact in a hierarchical manner.[3]
Subsumption architecture is a way of decomposing complicated intelligent behaviour into many "simple" behaviour modules, which are in turn organized into layers. Each layer implements a particular goal of the agent, and higher layers are increasingly more abstract. Each layer's goal subsumes that of the underlying layers, e.g. the decision to move forward by the eat-food layer takes into account the decision of the lowest obstacle-avoidance layer.
For example, a robot's lowest layer could be "avoid an object", on top of it would be the layer "wander around", which in turn lies under "explore the world". The top layer in such a case could be "create a map", which is the ultimate goal. Each of these horizontal layers accesses all of the sensor data and generates actions for the actuators — the main advantage is that separate tasks can suppress (or overrule) inputs or inhibit outputs. This way, the lowest layers can work like fast-adapting mechanisms (reflexes), while the higher layers control the main direction to be taken in order to achieve the overall goal. Feedback is given mainly through the environment.[14]
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.
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.

2. Robotic Control

2.1 Deliberative Architectures

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

2.2 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 is main drawback of reactive architectures.[10]

2.3 Programming Environments

To provide means for controlling of the Mindstorms robot that will be deployed within the RCX is a central assignment of this project. How will we implement programs that run on the RCX and what programming environment do we employ for this task?

2.3.1 Rcx Code

The first thing that comes to mind is to use RCX Code, the Mindstorms SDK, which is part of every robotics kit. Rcx Code is a proprietary graphical software developer's kit that is targeted at an audience not experienced in programming. Programs written with RCX Code are compiled into a sequence of opcodes and are then uploaded to the RCX via infrared. There are several disadvantages of that system though, the most significant ones being the following: unsatisfactory level of access to the internals of the RCX, no support of communication with the PC (upload only) and no way to program multi-threaded applications. Last but not least, one soon gets tired of assembling programs from graphical "command bricks" using the mouse.

2.3.2 Alternative RCX Programming

Within several weeks of the first release of Mindstorms, the hardware internals of the RCX were reverse engineered and made available publicly to the interested. A few months later, the open source community had produced a series of alternative ways to program the RCX, sometimes along with alternative firmwares. The most important of those are:

BrickOS, a C-based firmware which allows to write robot programs in standard C, using the standard GNU compiler tools. BrickOS gives complete low-level access to all of the RCX's internals. It is used mainly by system programmers who desire to take advantage of the last bit on the RCX.

NQC, a high level language with a C-like syntax. Comes with a compiler to produce opcode programs for the original LEGO firmware. NQC is the best established alternative programming system for the RCX. It is powerful yet easy to learn and not as complex as the original C-language.

LeJOS, a small Java Virtual Machine (JVM) implementation that runs on the RCX. This firmware gives users the chance to write control programs for robots in pure Java and comes with a fairly large API. LeJOS also includes a communications package that makes it very easy to establish a stream-based communication between PC and RCX.

Each of those high-level programming language implementations gives access to all of the RCX hardware and allows to write complex programs for the Hitachi 8300 micro controller. And since the purpose of this project is to analize the posibilities of development in this field there will be studied both the LeJOS, in the first part, NQC for the enhanced exploring robot and the BrickOS in the last part.[9,10]

LeJOS

LeJOS was originally conceived as TiniVM by Jose Solarzano who was challenged by the task of implementing a Java virtual machine for the RCX platform. The resulting interpreter had a size of about 10 kB which left about 18 kB RAM for Java programs and o®ered the following features (among others): Java on the RCX, preemptive threads, arrays, recursion, synchronization and exceptions. After the stable release of TinyVM the e®ort of completing and extending TinyVM was continued by other open source developers as the LeJOS project. The interpreter grew in size up to about 17 kB, almost inverting the ratio of program memory vs. firmware size.[10]

Tools

The LeJOS distribution comprises of a linker that produces a binary file from class files and a loader that performs the upload of this binary file to the RCX. An uploaded binary file is copied verbatim into memory on the H8300 and executed as soon as the user presses the ON button on the RCX brick.

NQC

NQC stands for Not Quite C, and is a language for programming several LEGO MINDSTORMS products. Some of the NQC features depend on which MINDSTORMS product you are using. This product is referred to as the target for NQC. Presently, NQC supports five different targets: RCX, RCX2 (an RCX running 2.0 firmware), CyberMaster, Scout, and Spybotics.

All of the targets have a bytecode interpreter (provided by LEGO) which can be used to execute programs. The NQC compiler translates a source program into LEGO bytecodes, which can then be executed on the target itself. Although the preprocessor and control structures of NQC are very similar to C, NQC is not a general purpose language - there are many restrictions that stem from limitations of the LEGO bytecode interpreter.

Types and Modes RCX

The sensor ports on the RCX are capable of interfacing to a variety of different sensors (other targets don't support configurable sensor types). It is up to the program to tell the RCX what kind of sensor is attached to each port. A sensor's type may be configured by calling SetSensorType. . There are four sensor types, each corresponding to a specific LEGO sensor. A fifth type (SENSOR_TYPE_NONE) can be used for reading the raw values of generic passive sensors. In general, a program should configure the type to match the actual sensor. If a sensor port is configured as the wrong type, the RCX may not be able to read it accurately.

```
SetSensor(SENSOR_1, SENSOR_TOUCH); // a touch sensor
SetSensor(SENSOR_3, SENSOR_TOUCH); // a touch sensor
SetSensor(SENSOR_2, SENSOR_LIGHT); // a light sensor
SetSensorMode(SENSOR_2,
SENSOR_MODE_RAW); // raw value from 0 to 1023
```

Navigation Implementation in Lejos

The RCX has a send message command that activates the IR port. To keep flashing the IR light while driving about, the rovers must a send message a couple of times each second. It doesn't matter what message we send, 0 or 255 , we have no reason to believe that it matters.

To detect reflecting obstacles we have to look for rapid changes in the light sensor reading. The best way to do this, is to sample the sensor quite often and compare two and two readings. If the difference between them is more than a set threshold we count that as an impulse _ the proximity detector has been tripped. This is very reliable, and experiment prove it detects almost anything from white walls to black boots or transparent soda bottles. And putting a small roof over the light-sensor to shield it from overhead light will even enhance the working range of the sensor a great deal.

This idea is implemented in a proximity library. In this library one small "driver task" periodically activate the IR transmitter and another small "driver task constantly polls the light sensor and checks for changes larger than some calibrated threshold value. If it finds such value it increases the global variable ProximityCounter. [9,10,12]

Below we have the run method of the ProximityDetector Thread. We first colect the value read by the light sensor, and then have the transciever send a packet of data and after a shor wait, we read the current value of the light sensor indication and compare it to our old value of light intensity. If the difference has become grater than the predefined proximity threshold, the listeners will be notified that an obstacle has been detected, and perform the coresponding actions for avoiding it.

```
public void run() {
    int oldValue;
    int newValue;
    // loop
    while(true) {
          // The old value read by the light sensor.
      oldValue = Sensor.S2.readValue();
      // IR trasciever send packet.
      Serial.sendPacket(packet, 0, 1);
      try {
          Thread.sleep(5);
      }catch(Exception e){}
      // Read new light sensor value.
      newValue = Sensor.S2.readValue();
      // Perform differencing
      int diff = Math.abs(oldValue - newValue);
      LCD.showNumber(diff);
      if(diff > proximityThreshold) {
          notifyListeners(diff);
      }
      try{Thread.sleep(160);}catch(Exception e){}
    }
  }
```

2.4 Learning

The robot learning problem is to design a robot so that it improves its performance through experience. To be precise, we must specify what performance and what experience are. Suppose the robot's performance is to be evaluated in terms of its ability to achieve some set of goals G. More precisely, suppose each goal is of the form $X \rightarrow Y : R$ where X and Y are both conditions representing a set of possible states and where R is some real valued reward. The goal $X \rightarrow Y : R$ is interpreted as if the robot finds itself in a state satisfying condition X then the goal of reaching a state satisfying condition Y becomes active, for which a reward R is received. For example_ the goal of recharging the battery when it is low can be represented in this way, by setting X to the sensory input "battery level is low", Y to the sensory input robot senses that it is electrically connected to the battery recharger and R to 100. Given a set of such goals we can define a quantitative measure of robot performance such as the proportion of times that the robot successfully achieves condition Y given that condition X has been encountered, or the sum of the rewards it receives over time. If we wish, we might further elaborate our measure to include the cost or delay of the actions leading from condition X to condition Y.

2.4.1 What and How to Learn

What and how should we design robots to learn. Two important dimensions along which approaches vary are the exact functions to be learned and the nature of the training information available. Here we consider a few possible learning approaches, then summarise some of the more significant dimensions of the space of possible approaches.

The most direct way to attack the robot learning problem is to learn the control function f directly from training examples corresponding to input-output pairs of f. Recall that f is a function of the form $f : S \rightarrow A$ where S is the perceived state and A is the chosen control action.[7, 15]

In some cases, training examples of the function f might not be directly available. Consider for example a robot with no human trainer. with only the ability to determine when the goals in its set G are satisfied and what reward is associated with achieving that goal. For example, in a navigation

task in which an initially invisible goal location is to be reached and in which the robot cannot exploit any gradients present in the environment for navigation, a sequence of many actions is needed before the task is accomplished. However if it has no external trainer to suggest the correct action as each intermediate state, its only training information will be the delayed reward it eventually achieves when the goal is satisfied. In this case it is not possible to learn the function f directly because no input-output pairs of f are available. An alternative approach that has been used successfully in this case is to learn a diferent.[16]

3. Experiment & Results

3.1 Evolving The Robot Controller Using Genetic Algorithms

In this section, we describe the final piece of our implementation - genetic algorithm on the robot controller 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 encoded (in binary) as 00, 01, 10 and 11 respectively. The robot's genotype (controller) is obtained by first ordering the Old State, Input pairs lexicographically. The robot's genotype is obtained by concatenating the corresponding Next State, Action . The Input value is 1 if the robot's proximity sensor senses an obstacle, 0 otherwise.

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 1: Parameters for Genetic Algorithm

Table 1 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).

3.2 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. 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.

4 Conclusion

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). *External factors include:*

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. We found that LegOS is a very powerful 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. There are also a couple of pitfalls in using legOS. Firstly, legOS has a priority inversion problem 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.

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 Learing. *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, .S., Martin, A., Robinson, G.E., Poppy, G.M., Riley, J.R. *Ontogeny of orientation flight in the honeybe 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. ttp://www.handyboard.com.

[18] The MIT Programmable Brick Project. http://el.www.media.mit.edu/groups/el/-projects/programmable-brick/.