Work In Progress - Programming in a Confined Space – A Case Study in Porting Modern Robot Software to an Antique Platform

Stacey L. Montresor, Jennifer S. Kay, Michel Tokic, Jonathan M. Summerton
smontronj@gmail.com, kay@elvis.rowan.edu,
tokicm@hs-weingarten.de, summer74@students.rowan.edu

Abstract - **In a typical introductory AI class, the topic of reinforcement learning may be allocated only a few hours of class time. One engaging example of reinforcement learning uses a crawling robot that learns to use its two-degree-of-freedom arm to drag itself forward. Unfortunately, the cost of the required hardware is prohibitively expensive for many departments for what is typically a once-a-semester demonstration. So we decided to port the algorithm to a platform that many departments may already have on hand: the LEGO Mindstorms RCX 2.0. Initially the task seemed relatively straightforward: build a robot base out of LEGO parts and implement the algorithm in the Not Quite C language. However the challenges of designing a robot arm without servos and attempting to trim code down to a size that would fit on the RCX has proven to be as educational to the undergraduates working on the project as we hope the final product will be to students in AI classes. This paper describes the challenges we have faced and the solutions we have implemented, as well as the work that remains to be completed.**

Index Terms –Artificial Intelligence, Computer Science Education, crawling robot, LEGO Mindstorms RCX 2.0, Not Quite C (NQC), reinforcement learning, robotics

INTRODUCTION

In a typical introductory AI class, the topic of reinforcement learning [1] may be allocated only a few hours of class time. Kimura et. al [2] and Tokic et. al [3] developed an engaging example of reinforcement learning using a crawling robot that learns to use its two-degree-of-freedom arm to drag itself forward. The robot learns to move forward using value iteration or Q-learning [1] over the span of roughly 15-20 seconds. The system is particularly attractive pedagogically because it incorporates a visualization tool that shows the two dimensional state space in which the robot actions are moves to neighboring states in the two dimensional grid world.

Appealing as this system is, the expense of purchasing the specialized robot platform is beyond the means of many computer science departments for such limited use. We decided to port the algorithm to a platform that is inexpensive and that many departments may already have on hand: the LEGO Mindstorms RCX 2.0 [4][5]. This is from the first generation of LEGO robotic kits, and has subsequently been supplanted by the NXT.

Initially the task seemed relatively straightforward: build a robot base out of LEGO parts supplied with the RCX 2.0 kit and implement the algorithm in the Not Quite C (NQC) language [6][7]. However the challenges of designing a robot arm without servos and attempting to trim code down to a size that would fit on the RCX has proven to be as educational to the Computer Science undergraduate students working on the project as we hope the final product will be to students in AI classes.

ROBOT HARDWARE SETUP

Our LEGO Robot was built in the image of Tokic's crawling robot (Figure 1). The LEGO robot is controlled by the RCX 2.0 brick [4][5] which is mounted on top of the robot. The two arm joints are driven by 9v motors. The first joint connects the arm to the base and the second joint connects the "claw" to the arm.

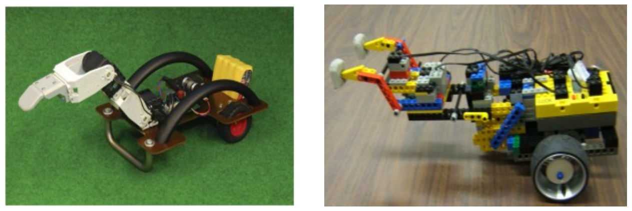
FIGURE 1: TOKIC'S ORIGINAL CRAWLER AND OUR LEGO RCX VERSION

The original RCX sets only provide motors that can be moved for a particular amount of time, rather than to a specified angle. Thus we needed to construct a way to sense the position of the two joints, as well as to determine the movement of the robot's single axle wheel. Our first plan was to supplement the LEGO parts available in the standard RCX kits with three LEGO rotation sensors (LEGO #9756) which report changes in rotation rather than absolute position, but we still needed a way to ensure that the two joints began in the correct initial positions. We added two touch sensor bricks (already included with the kit) and initialize the joints by moving them into their uppermost positions until the touch sensors are triggered, ensuring that the start position is always the same. But then we encountered the first (of many) limitations we would have to overcome because of our choice of platform: the RCX 2.0 brick has three input connections. While touch sensors can be combined on a single input connection, the rotation sensors cannot and we had to eliminate one rotation sensor. Further evaluation of the joints showed that the effect of the force on the claw was harder to predict than the constant weight of the arm, therefore we opted to keep the rotation sensor on the claw. However, over time our estimate of the positions of both joints still degrades. To alleviate this, whenever the joints are placed in the first or second position, we reinitialize using the corresponding touch sensor.

CODING IN A CONFINED SPACE

Since our ultimate goal is to create a classroom demonstration that can be built with hardware that many schools have on hand or can acquire cheaply, we continue to believe that our choice of the LEGO RCX platform is a good one. Nonetheless, compressing the code down to a size that will fit into its 16K of RAM has been a challenge. NQC allows only 8 “subroutines” which provide a single copy of code shared between different callers, but do not have arguments. There is no limit to the number of inline "functions” used other than the overall code size must be within the 16K limit [6].

We implemented the value-iteration learning algorithm as used on the robot in [3] as closely as the RCX 2.0 brick would allow. To ensure our understanding of the value-iteration algorithm, we first simulated the algorithm using Java using the same reward model as in [3] and confirmed that our implementation matched the original by successfully regenerating the same values with our simulation.

The next step seemed simple, code a few methods to handle the movements and re-implement the simulated algorithm using NQC. At this point, it became necessary to scale down the state-space model used in [3] from 5x5 to 3x3. The RCX 2.0 brick limits us to the use of 32 global variables and 16 local variables [6]. To implement a 5x5 state space requires 125 variables to store the state and reward values. By reducing the state space to a 3x3 grid we maintain a large enough space to include a subset that would represent an optimal cycle while requiring only 33 variables to store the state and reward values. Nonetheless, in addition to the variables needed to store the current state, we needed to store information about the action to be completed and the last sensor reading, and so we needed a total of 36 global variables, still more than we had available.

NQC variables are 16-bit signed integers. Using bit manipulation we were able to combine three reward values into a single variable and we use this technique to store the left/right reward values. Though this frees up the total number of variables needed, the additional lines of code to implement the compression and uncompression algorithm add to our RAM consumption.

We initially coded the joint movements using three methods to execute the actual movements to be made. Within these methods, the compression function needed to be called which prevented them from being “subroutines” since in NQC subroutines cannot call other subroutines! To reduce the memory usage resulting from inline functions, these three methods were broken down into 24 functions representing individual moves from one state to the next.

Due to the lack of variables, we are not able to store individual state’s policies. Instead, during exploitation we iterate through all neighboring states and comparing the possible reward values, choose the action that result in the greatest reward value.

MODIFYING THE LEARNING ALGORITHM

The discounting factor, γ, of the value-iteration algorithm has a range $0 < \gamma < 1$. Tokic et. al’s experiments [3] resulted in a recommended discounting factor of 0.9 for short learning times finding the optimal cycle. Since the RCX 2.0 brick is restricted to the use of integers only, we are forced to use a value of 1 for the discounting factor. While setting the discounting factor to 1 removes the theoretical guarantee that the value iteration algorithm will converge, we have demonstrated in simulation that the robot can learn a potentially suboptimal forward-walking policy with a discounting factor of 1.

We used a simple ε-Greedy technique to regulate exploration and exploitation. This is accomplished by randomly choosing a value between [0, 1] and comparing it to ε, which is set to a high value initially and decreased throughout the learning process. Restricted to the use of integers, we randomly choose values from [0, 500]. We set ε to 500 at the start of the learning process. After an action is completed we decrease ε by 10 similar to [3]. Alternatively, instead of decreasing the value of epsilon, a constant value of epsilon (e.g. epsilon=50) may also be used.

CURRENT STATUS

Our hardware design is complete, and we have demonstrated that our robot can “walk” forward when we hardcode the sequence of states in our 3x3 grid to be visited. After much code manipulation, our version of the code is small enough to be loaded into the RCX 2.0. In our current implementation, the robot randomly explores and exploits throughout the 9 possible states and builds the associated reward table. The resulting values can be viewed through the Bricx Command Center. We are currently experimenting with various values for ε that will result in at least a successful cycle, if not the optimal cycle, for consistently learning to successfully walk forward.

FUTURE WORK

For classroom use, we would like to transmit the data related to the states throughout the process into a form that is accessible to all students that will allow them to evaluate the steps in determining the optimal cycle. We are also developing a full web site that will provide step-by-step instructions on how to replicate our system so that others may use it in their AI classes.

REFERENCES

[1] Sutton, R. and Barto, A. 1998. Reinforcement Learning: An Introduction. Cambridge, MA: MIT Press.

[2] Kimura, H., Miyazaki, K. and Kobayashi, S. 1997. Reinforcement Learning in POMDPs with Function Approximation. In *ICML '97: Proceedings of the Fourteenth International Conference on Machine Learning*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 152-160.

[3] Tokic, Michel, Ertel, Wolfgang, and Fessler, Joachin, "The Crawler, A Class Room Demonstrator for Reinforcement Learning", *Proceedings of 22nd International FLAIRS Conference* 2009, AAAI Press, pp. 160-165.

[4] "LEGO Education: Store: RCX Programmable Brick," http://www.legoeducation.us/store/detail.aspx?ID=336&bhcp=1, accessed March 2010.

[5] "Lego Mindstorms," http://en.wikipedia.org/wiki/Lego_Mindstorms, accessed March 2010.

[6] Baum, Dave, *NQC Programmer's Guide*, Version 2.5 a4 http://neuron.eng.wayne.edu/LEGO_ROBOTICS/nqc_guide.pdf accessed October 2010.

[7] "NQC – Not Quite C," http://bricxcc.sourceforge.net/nqc/, accessed March 2010.

AUTHOR INFORMATION

Stacey Montresor, Undergraduate Computer Science Student, Rowan University, USA, smontronj@gmail.com

Jennifer Kay, Associate Professor, Computer Science Department, Rowan University, USA, kay@rowan.edu

Michel Tokic, Research Assistant, Laboratory of Autonomous Mobile Servicerobots (ZAFH AMSER), University of Applied Sciences Ravensburg-Weingarten, Germany, tokicm@hs-weingarten.de

Jonathan Summerton, Undergraduate Computer Science Student, Rowan University, USA, summer74@students.rowan.edu