



**University  
of Surrey**

**Department of Computing**

Combining heuristics and Q-learning in  
an adaptive light seeking  
robot

Constantinos A. Kroustis,  
Matthew C. Casey

January 2008

Computing  
Sciences  
Report

**CS-08-01**

# Combining Heuristics and Q-Learning in an Adaptive Light Seeking Robot

Constantinos A. Kroustis, and Matthew C. Casey

**Abstract**—The use of adaptive techniques in robots that operate in a real environment is an important area of research. In particular, with our developing understanding of brain function, robots can be used to explore biologically motivated algorithms. However, developing a framework in which such algorithms can be tested is a challenge, requiring low-level processing functions that can take care of sensory pre-processing and reflex motor actions for adaptive algorithms. In this paper, we present a framework that combines together low-level heuristics (reflex actions) and a high-level adaptive algorithm (Q-Learning). We implement this framework on the LEGO Mindstorms NXT. The robot is designed to learn how to find a light source in an unknown environment. Our results show that the combination of the heuristic with the adaptive algorithm reduce the number of execution steps required to find the light, and that the framework enables the adaptive algorithm to successfully complete its task. Through this implementation, we also demonstrate how the NXT can be used as a suitable platform for the development of complex algorithms, using remote commands via Bluetooth.

## I. INTRODUCTION

THE use of adaptive techniques in robots that operate in a real environment is an important area of research. Tackling real-world tasks is an interesting challenge, not least because it allows us to explore how carefully constructed algorithms operate in a dynamic environment (cf. [5]), but also because it allows us to evaluate models of adaptive behavior motivated by our understanding of brain function in animals. However, developing robots that can operate in a real environment and adapt to their surroundings via sensor inputs provides both physical engineering and ease of development problems. In addition, whilst we have a developing understanding of specific brain functions involved in the sensory-motor loop (cf. the superior colliculus in mammals [4]), translating the functionality of such isolated behavior into a robot requires us to provide a sufficient framework in which these models can be tested. For an adaptive algorithm, this means that the framework must provide the conditions under which the algorithm can learn from sensory inputs to evaluate behavior, rather than to attempt to model lower level functionality and hence increase complexity. This in itself is

a challenge in terms of dealing with sensory noise [5], and with the unexpected, all in an environment without excessive constraints.

The aim of this paper is twofold. First, in simple terms we will explore how adaptive algorithms that implement a sensory-motor loop can be embedded into a robot that provides the necessary framework in which the algorithm can operate. To provide an appropriate framework, we use a modified form of Brooks' subsumption architecture [2] to model different layers of behavior. To enable the higher-level adaptive algorithm to operate effectively, we implement a simple heuristic in the lower levels so that the learning algorithm is only trained on appropriate sensory inputs. For simplicity, we chose Q-learning [14] as our adaptive algorithm.

Second, with the advent of low-cost, flexible robot kits, such as LEGO Mindstorms [12], which offer the capability for rapid prototyping of ideas, as well as supporting high-level programming languages [6], we explore how our framework can be developed with the new NXT system, exploiting increased sensory, processor and communication capabilities. We hope that our experiences of developing adaptive robots within this environment provides a timely exploration of this new technology that makes robot development available to a wider audience, such as for educational institutions using these technologies to teach programming [1] and artificial intelligence [10].

In section II of this paper, we explore the framework in which we combine Q-learning and heuristics. In section III we review the implementation of the framework within the NXT, evaluating approaches to successfully allow the robot to adapt given the physical robot constraints. In section IV we evaluate the developed robot with a series of experiments and discuss how these achieve our aims. In section V we conclude and discuss future work.

## II. COMBINING HEURISTICS AND Q-LEARNING

A number of machine learning techniques have been used in robotics, including supervised [9], unsupervised [15], reinforcement [5] and evolutionary techniques [16]. The use of evolutionary learning requires the simulation of a large number of scenarios prior to installation in the robot in order to evolve an appropriate behavior. Supervised techniques can suffer from the same problem because of the need to have a target response to generate an error. In contrast, unsupervised learning is more appropriate for real-time

Manuscript received September 14, 2007.

C. A. Kroustis is with the Department of Computing, University of Surrey, Guildford, Surrey, GU2 7XH, UK (e-mail: coskrousti@hotmail.co.uk).

M. C. Casey is with the Department of Computing, University of Surrey, Guildford, Surrey, GU2 7XH, UK (corresponding author phone: +44 (0) 1483 689635; fax: +44 (0) 1483 686051; e-mail: m.casey@surrey.ac.uk).

adaptation as it can adapt using sensory inputs alone. Such techniques are also more biologically plausible, allowing us to test models of brain function, such as the development of sensory topographic maps in the superior colliculus [11]. Reinforcement learning provides a simple compromise between learning on sensory inputs only versus learning using a desired response. Algorithms, such as Q-learning [14], can be used to map sensory input states to actions using a goal [3]. Although Q-learning is not biologically plausible, the use of goal driven learning is. For example, in the superior colliculus (SC), activity in topographic maps for coincident visual, auditory or somatosensory signals is enhanced by cortical feedback, strengthening the connection between spatially and temporally located events [11].

Using models of brain function in robots may offer advantages. For example, a model of the SC in a robot may be able to learn how to react to different sensory inputs and their combination. However, to test such a model relies upon an appropriate framework in which to situate the model. To continue with the example of the SC, in the first instance, the animal has a series of lower level behaviors that operate autonomously (sensory-motor reflex actions), sensory processing and motor output functions (sub-cortical), and higher level coordinating activities (cortical). By operating within this framework, pre-processed sensory signals are input to the SC, with a resulting motor output (eye movement), all mediated by the cortex. From a goal-driven perspective, the cortical feedback only operates when it is appropriate to do so (cf. linking together visual and audio stimuli when they are coincident), which implies a lower level of autonomous behavior that operates whenever the adaptive algorithm is not.

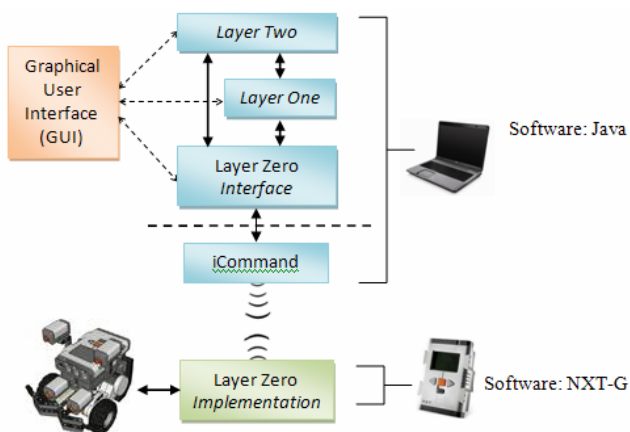


Fig. 1. The Structure of the controller. *Layer n* can invoke methods only on *Layers n-1...0*. *Layers One* and *Two* are implemented on the computer using Java, whilst *Layer Zero* is implemented on the robot using the standard programming software for the NXT.

This simplified notion leads us to the implementation of three layers: low level sensory and motor processing, sub-cortical autonomous operation, and the adaptive algorithm. To implement such a layered approach we use a modified version of Brooks' subsumption architecture [2] (see Fig. 1).

At the lowest level *Layer Zero*, is the robot's sensor and motor operations. In *Layer One*, we implement an autonomous sensory-motor loop heuristic that aims, through a simple set of rules, to put the robot into a situation in which the top layer can operate and learn. In *Layer Two*, we implement our adaptive algorithm, in this instance Q-learning.

To explore this framework, we chose the simple goal of the robot seeking out a single light source. This task will be carried out within an unknown environment with a single light source and the potential for obstacles. In this environment, the *Layer One* autonomous heuristic is designed to randomly move the robot until it detects a minimum level of light, at which point it hands control to the adaptive algorithm, therefore providing the learning process with a series of appropriate inputs. Here, Q-learning is used to map sensory states to actions. If at any time the light level falls below the required minimum, then the heuristic takes control once again.

### III. IMPLEMENTATION USING NXT

For our implementation, we used the LEGO Mindstorms NXT kit [12], which is the latest robotics toolset provided by LEGO. This kit provides an easy and robust way for building and programming robots. One of the most powerful features of this kit is the support for different programming techniques (such as Java, C#, Perl), in addition to the proprietary LEGO software developed by National Instruments [8]. The kit supports programs running entirely on the NXT, as well as those running remotely, communicating with the NXT brick using the open source LEGO Communication protocol over Bluetooth [13].

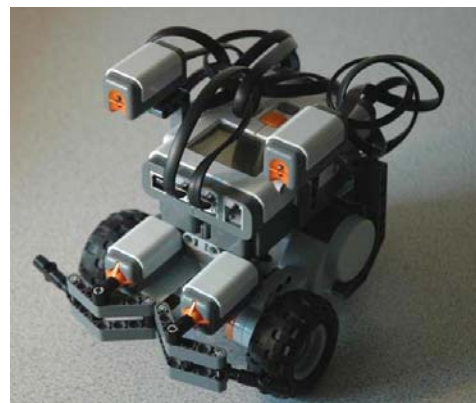


Fig. 2. The LEGO Mindstorms NXT robot used for this project. The two light sensors are located at the top in the front and the two touch sensors in the front at the bottom.

The robot consisted of a modified version of the basic design that is described in the Quickstart Guide of the NXT kit [12]. It includes two light sensors facing forward, located on the top left and right corners and two touch sensors (bumpers) located at the lower left and right corners (see Fig. 2). Movement is achieved using two independently controlled motors that rotate each wheel. This gives the

robot the ability to rotate within its own footprint (turn on one spot). In particular, when the robot turns, both wheels are rotating (in opposite directions), ensuring that the angle of the robot is changed while its position remains the same. Full construction details, source code and further information on the implementation can be found at <http://nxt-controller.freehostia.com/>.

Even with the 320 KB of memory that the NXT supports, in order to implement both the heuristic and the adaptive algorithm for the robot, it was clear early in the design that the NXT's memory would be insufficient for this program. However, with the program running entirely on the computer the Bluetooth link became a bottleneck. This can happen during any kind of motor movement when the rotation sensors are being queried to determine the amount of movement actually undertaken. This problem was addressed by distributing the processing, allowing part of the controller to run on the NXT and the other part to run on the computer. The standard NXT firmware and LEGO software (NXT-G) was used to program the brick to respond to Bluetooth commands, whilst the iCommand Java package, release 0.5 [7], was used on the controlling computer in order to send and receive data through the Bluetooth link. Fig. 1 shows the processing distribution for each layer.

#### A. Layer Zero: Low Level Navigation

Layer zero is responsible for the implementation of all low level navigation mechanisms and reflex actions. Using iCommand, this layer is split between the computer and the NXT. The methods provided by this layer are:

- setDirection(x): rotate  $x^\circ$  to the left or to the right.
- goToDirection(x): rotate to  $x^\circ$  regardless of current direction.
- goForwardFull(x): travel forward x cm.
- getState(): get the current state of the robot, including readings from the light sensors, bumper status and estimated coordinates of the robot's position.

All except the getState() method return immediately. The getState() method blocks execution until the required feedback is received by the computer.

Overriding all other functionality, this layer reacts to obstacle collisions by reversing for a short distance (approximately 15cm). No turn is made to avoid the obstacle, but the collision is reported as part of the getState() response.

#### B. Layer One: Exploration Heuristic

This layer gives the robot the ability to plot its place in the environment by recording the coordinates of visited locations. Each time the getState() command is executed this layer receives the coordinates of the robot's new position in order to update its internal map. The method which implements the developed heuristic for the robot is the goRandom() method. When invoked, this method enables the robot to wander around randomly exploring new

```

LOOP(3)
{
  Turn 30° in the opposite direction of the last turn taken by Layer Two.
  Ask Layer Zero for robot's state.
  IF(Light>low bound) THEN Control returns to Layer Two
  ELSE CONTINUE
}

LOOP( $\infty$ )
{
  Ask Layer Zero for robot's state.
  IF(Repetitions=15) Then Travel to the least visited area.
  ELSE
  {
    IF(Light>low bound) THEN Control returns to Layer Two

    IF(Left bumper pressed) THEN Direction=right
    ELSE IF(Right bumper pressed) THEN Direction=left
    ELSE IF(Both bumpers pressed) THEN Direction=same
    ELSE IF(No bumpers pressed) THEN Direction=Direction of bright
light

    IF(Last time a bumper was pressed AND Now the opposite bumper is pressed)
THEN Angle=180°
    ELSE Angle=RANDOM SELECTION(30°,40°,50°,60°,70°,90°)

    Distance=RANDOM SELECTION(30cm,40cm,50cm)

    Turn according to Direction and Angle and travel forward according to
Distance.
  }
}

```

Fig. 3. The implemented heuristic for the robot. This was developed through evaluating the ability of the robot to explore unknown environments.

areas (see Fig. 3.). In order to make exploration more efficient a set of rules is used which enables the robot to react more intelligently in the presence of obstacles. The goRandom() method is invoked from *Layer Two* when the light detected is below a predefined threshold value. Control returns back to *Layer Two* as soon as this method finishes execution, which is as soon as the light detected gets above that threshold.

The main factors that influence the movement initiated from the heuristic are the light values and the presence of obstacles. In particular, the heuristic forces the robot to turn to the direction of the highest light value. As described earlier, *Layer One* uses the methods provided from *Layer Zero* in order to initiate movement as well as monitor the state of the robot.

#### C. Layer Two: Q-learning

This layer is the source of control for the application, meaning that lower layers cannot initiate any action on their own. All method invocation decisions come from here. This layer operates in three different states according to the amount of light detected by the light sensors:

- 1) If the maximum value for either light sensor is above the required higher threshold, the program terminates (the light source is found).
- 2) If the maximum value for either light sensor is below the specified lower threshold, then the goRandom() method of *Layer One* is invoked.

- 3) If the maximum value for either light sensor is between the lower and higher thresholds, then Q-learning is invoked.

Here, Q-learning is implemented as shown in Fig. 4, where:

- $s$  is the current state of the robot.
- $a$  is the last action taken.
- $s'$  is the new state.
- $a'$  is the action with the maximum value for state  $s'$ .
- $\gamma$  is the reward-discount parameter ( $0 \leq \gamma \leq 1$ ).
- $r$  is the reward received.
- $B$  is the learning rate ( $0 \leq \beta \leq 1$ ).

```

INITIALIZE  $Q(s,a)$  with random values
WHILE ( $s$  is not terminal)
{
  Select action  $a$  from the current state  $s$  using the selection
  policy (e.g.,  $\epsilon$ -greedy)
  Execute action  $a$  and observe reward  $r$  at new state  $s'$ 
   $Q(s,a) = Q(s,a) + \beta[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$ 
   $s = s'$ 
}

```

Fig. 4. The Q-learning algorithm [14].

One issue in the implementation of this algorithm is keeping a reasonable balance between exploring the environment for new training samples and using what is already learned from the operation so far. The selection policy used here is the  $\epsilon$ -greedy exploration policy [5]. According to this method, actions are selected using a constant  $\epsilon$  (exploration factor). Specifically, an action is selected randomly (exploration) with probability  $\epsilon$  and according to the look-up table (exploitation) with probability  $1 - \epsilon$ .

TABLE I  
Q-LEARNING STATES

State	Left and Right Light Sensor Readings	Bumper Status
0	Left $\approx$ Right	None Pressed
1	Left $>$ Right	
2	Left $<$ Right	
3	Left $\approx$ Right	Left Pressed
4	Left $>$ Right	
5	Left $<$ Right	
6	Left $\approx$ Right	Right Pressed
7	Left $>$ Right	
8	Left $<$ Right	
9	Left $\approx$ Right	Both Pressed
10	Left $>$ Right	
11	Left $<$ Right	

Left  $\approx$  Right is equivalent to  $|\text{Left} - \text{Right}| \leq \delta$ , where  $\delta$  is a specified tolerance constant (4 in this instance).

All possible states that Q-learning can identify belong to a finite set of uniquely identifiable states. Since the available sensors on the robot are light and touch sensors, the description of each state is a combination of their values as shown in Table I. The set of possible actions used by the learning algorithm is shown in Table II.

The actions set was developed in order to establish actions that could “assist” the robot in moving towards the light source and avoid obstacles, without being exhaustive. The most significant criteria used in this process were the following:

1. Balance between the number of actions of each type.
2. Provide a variety of rotation angles.
3. Provide a variety of distances for forward movement.

The second criterion is essential since large rotation angles are useful in avoiding obstacles whereas small angles are vital in locating the light source.

In order to learn, the reward  $r$  is determined to be proportional to the change in light sensed between the last and current states. Avoiding obstacles is also rewarded.

TABLE II  
Q-LEARNING ACTIONS

Action	Description	Type
0	Left 15°	Rotation
1	Right 15°	
2	Left 20°	
3	Right 20°	
4	Left 40°	
5	Right 40°	Forward Movement
6	Forward 15 cm	
7	Forward 20 cm	
8	Forward 25 cm	

#### IV. EXPERIMENTS AND EVALUATION

The experiments required setting the robot to operate in a dark room with a single light source (normal desktop lamp). In each experiment the robot was left to operate without any guidance until it found and faced the light source. At that point execution was automatically terminated (*Layer Two* stops execution when the light source is found). Each experiment was repeated a number of times under the same conditions. The threshold values for the light detected were 50 (low light threshold) and 85 (high light threshold), since the values returned from the lights sensors are in the range of 0-100. The values for the configuration parameters of Q-learning were the following:

- *exploration factor* = 0.25: the robot relies more on the values of the Q-learning table.
- *learning rate* = 0.3: a small value was sufficient since there was no need for high convergence speed.
- *Reward discount* = 1: the maximum value for the reward is obtained.

##### A. Experiment 1: Evaluating Q-learning

The purpose of the first experiment was to determine if the robot could learn to move towards the light source in a simple environment. Since the aim was to evaluate only the performance of the learning method (*Layer Two*), it was necessary to minimize the time that the robot would spend moving randomly (*Layer One* in control). This would only be possible if the light detected by the robot would be above

the lower threshold. This was accomplished by keeping the dimensions of the environment small (2 x 1.3m) and setting the robot to face the light source at its starting position (Fig. 5). This experiment was repeated 10 times.

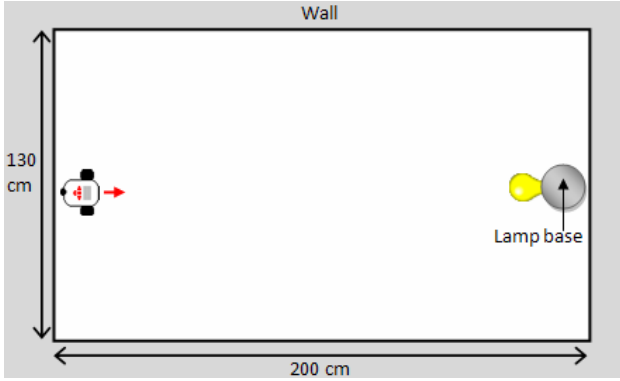


Fig. 5. Experiment 1 environment.

During all runs, the robot was able to locate and face the light source. Over all runs, the robot found the light in an average 44 execution steps.

To demonstrate the learning process, Fig. 6 shows the values of the Q-learning table for the first run against the actions for state 8 at three different execution steps. As we can see, the robot learns that the best action to execute when the left light is brighter than the right and no bumpers are pressed is to turn left 20°. Also, as operation proceeds the selected action becomes more and more preferable while the values for the other actions are decreased.

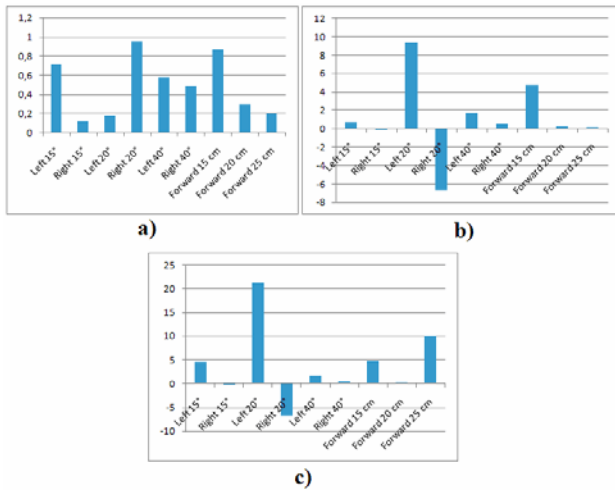


Fig. 6. Instances of the learning process for state 8 (Left < Right and right bumper pressed) in the first run of experiment 1. These represent the values of the Q-learning table at execution steps a) 20; b) 40 and c) 55.

Whilst this demonstrates the ability of the learning process to associate appropriate actions to states, we can also note that the learning also results in a preference to turn rather than move forward. For example, in the first run of the experiment only 16 of the 55 steps were forward movements (actions 6-8) and in the second run only 11 of 43. This explains the relatively large number of actions

taken considering the small area which was covered. The study of the execution history showed that the robot appears to be insufficiently motivated to move forward, since light values change just by rotating left or right. During the early stages of the learning process, forward movement is initiated when the algorithm selects an action randomly. However, as learning progresses execution of forward movement becomes more probable, if the feedback received from previous executions is positive.

Since the Q-learning state table includes all possible states, it is likely during any given run that not all states will be encountered. For example, the robot may never come across a situation where both bumpers are pressed and the light on the left is less than the light on the right side. In particular, during this experiment, only states 0, 1, 2, 5, 7 and 8 were encountered, with state 2 being active 81.5% of the time on average. The reason for this was the error in the robot's movement. Specifically, when the robot is moving forward it tends to follow an arched trajectory rather than a straight line (possibly due to the carpeted flooring). This error made the robot approach the light source always from the left, thus putting the robot in state 2 (Left Light < Right Light and no bumpers pressed). This example demonstrates that the learning algorithm is capable of overcoming such anomalies in unknown environments, whilst still finding the light source.

#### B. Experiment 2: Evaluating the Heuristic

In the second experiment the aim was to evaluate the performance of the heuristic (*Layer One*). To achieve this, the same environment as Experiment 1 was used, but the robot was positioned to start facing a dark corner (as shown in Fig. 7). This was done in order to limit the amount of the light detected at the robot's starting position, and hence to initiate *Layer One* operation. This experiment was divided into two phases. During the first phase the robot was set to operate as previously explained and the experiment was repeated 10 times. During the second phase the same procedure was repeated but this time the heuristic was not allowed to operate leaving only Q-learning (*Layer Two*) responsible for navigating the robot towards the light source. This was done in order to identify if the heuristic was in fact able to minimize the execution steps required to locate and face the light source.

On average, by initiating the heuristic in *Layer One*, the robot found the light in 30.1 steps (27.6% of the moves were spent in the heuristic and the rest on Q-learning), far less than the average number of the steps required (177.7) when only Q-learning was used. One of the heuristic's features that proved to be very helpful was that when in two subsequent steps both bumpers are pressed, the robot is instructed to turn 180°. This feature allowed the robot to successfully escape from corners.

#### C. Experiment 3: Avoiding Obstacles

The purpose of this experiment was to see how the robot

would react to the addition of an obstacle. The reason for adding the obstacle in the center of the environment was to interrupt the route that was used by the robot most often (Fig. 8). The robot was started facing one of the corners, as before. This experiment was repeated 5 times.

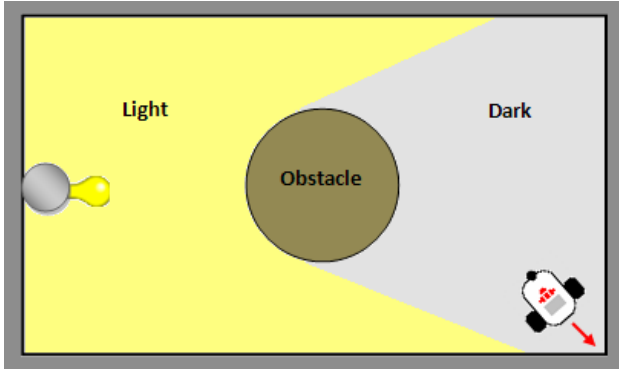


Fig. 8. Experiment 3 environment.

On average the robot found the light in 29.4 steps, with 43.9% of the moves spent in the heuristic. The purpose of this experiment was to observe if the robot could find its way through the narrow openings between the obstacle and the walls. However, the addition of the obstacle also blocked the light, creating a large dark area around the starting position of the robot. As a result, the presence of the obstacle itself did not cause any difference in the robot's behavior, which managed to avoid the obstacle appropriately, but the large dark section meant that the heuristic was in operation for longer than expected.

#### D. Discussion

In general, the results from the experiments indicate that the primary objective of the controller has been achieved. Specifically, the robot can adapt to an unknown environment in order to locate a light source, even when the robot appears less motivated to move forward. Furthermore, the heuristic has had a positive effect on the performance of the controller, significantly reducing the number of steps required to locate and face the light.

Some increase in required execution steps was caused by anomalies in the intensity of the light reflected from the surrounding objects. In an ideal environment, the intensity of the reflections would increase with proximity to the light source. However, in this non-ideal environment, reflections varied due to the different types of materials in the room. The results show that the controller depends on these reflections, such that if the intensity increases closer to the source, then it is easy for the robot to succeed. This aspect of the robot's behavior is confirmed by the last experiment where the robot need longer after becoming trapped for a short time in the darker regions.

Comparing the developed framework to other related work, such as [5], we can argue that far more complex behavior has been achieved by combining a heuristic with the adaptive algorithm. This was a key aim of this work, and shows that the framework can provide a model in which

biologically inspired adaptive algorithms can be tested, with a suitable processing architecture to process sensory stimuli and reflex actions.

#### V. CONCLUSION

In this paper we have presented a controller for an adaptive robot which successfully tackles the real-world task of light seeking. The robot can operate in a real environment and adapt to its surroundings using only sensor inputs.

The controller itself can be used as a framework for testing other behavioral models, since it provides an environment where an adaptive algorithm can learn from sensory inputs to evaluate behavior without having to model lower level functionality, ideal for more advanced biologically inspired robotics.

The development of this controller using the LEGO Mindstorms NXT toolset has demonstrated how this system can be used in the implementation of more complex applications by exploiting the Bluetooth communications.

#### REFERENCES

- [1] D.J.Barnes, "Teaching Introductory Java through LEGO MINDSTORMS Models," in *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, 2005, pp. 147-151.
- [2] R.A.Brooks, "A Robust Layered Control Systems for a Mobile Robot," *IEEE Journal of Robotics and Automation*, vol. 2, pp. 14-23, 1986.
- [3] A.Burkov and B.Chaib-draa, "Adaptive Play Q-Learning with Initial Heuristic Approximation," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2007, pp. 1749-1754.
- [4] G.A.Calvert, C.Spence and B.E.Stein, *The Handbook of Multisensory Processes*. Cambridge, MA.: A Bradford Book, MIT Press, 2004.
- [5] K.Framling, "Adaptive Robot Learning in a Non-stationary Environment," in *Proceedings of the 13th European Symposium on Artificial Neural Networks*, 2005, pp. 381-386.
- [6] S.Hassenplug, "NXT Programming Software," <http://www.teamhasenplug.org/NXT/NXTSoftware.html>, 2007.
- [7] LeJOS, "LeJOS, Java for LEGO Mindstorms," <http://lejos.sourceforge.net/index.php>, 2005.
- [8] N.I.Corporation, "How LEGO MINDSTORMS NXT Works," <http://www.ni.com/academic/mindstorms/works.htm>, 2007.
- [9] H.G.Park and S.Y.Oh, "A Neural Network Based Real-Time Robot Tracking Controller Using Position Sensitive Detectors," in *Proceedings of the IEEE International Conference on Neural Networks*, 1994, pp. 2754-2758.
- [10] S.Parsons and E.Sklar, "Teaching AI using LEGO Mindstorms," in *Proceedings of the AAAI Spring Symposium on Accessible Hands-on Artificial Intelligence and Robotics Education*, 2004.
- [11] B.E.Stein and M.A.Meredith, *The Merging of the Senses*. Cambridge, MA.: A Bradford Book, MIT Press, 1993.
- [12] The LEGO Group, "LEGO.com MINDSTORMS NXT," <http://mindstorms.lego.com/>, 2007.
- [13] The LEGO Group, "LEGO.com MINDSTORMS Overview," <http://mindstorms.lego.com/Overview/NXTreme.aspx>, 2007.
- [14] C.J.C.H.Watkins and P.Dayan, "Q-Learning," *Machine Learning*, vol. 8, pp. 279-292, 1992.
- [15] E.Zalama, P.Gaudiano and J.Lopez-Coronado, "A Real-time Unsupervised Neural Network for the Control of a Mobile Robot," in *Proceedings of the IEEE International Conference on Neural Networks*, 1994, pp. 2848-2853.
- [16] B.-T.Zhang and S.-H.Kim, "An Evolutionary Method for Active Learning of Mobile Robot Path Planning," in *Proceedings of the International Symposium on Computational Intelligence in Robotics and Automation*, 1997, pp. 312-317.