**LeJOS Study:**

**Find the Dark**

**CS491**

**Special Topics (AI)**

March 31, 2015

By Cody Kubicek

And Tyler Rusch

**Abstract:**

This study serves as a look into full implementation of an algorithm. When merely testing code, all inputs and outputs are consistent and perfect—if no errors exist within the code, the result will be predictable and accurate every time. However, when switching to physical implementation, this is not always the case. Inputs may be flawed and execution may be less than optimal, and it is important to see how obtaining a degree of robustness can require a different approach.

Contents

[1. Introduction 1](#_Toc412577616)

[2. Background 1](#_Toc412577617)

[3. Approach 2](#_Toc412577618)

[4. Results 7](#_Toc412577619)

[5. Conclusion 8](#_Toc412577620)

[6. Appendices 9](#_Toc412577621)

List of Figures

Figure 1. Simple Reflex Agents…………….……………………………………………………….…1

Figure 2. The Robot……………………………………………………………………………………..3

Figure 3. Robot Scanning Path……………….………………………………………………………..4

Figure 4. The Moderately Populated Table….………………………………………………………..6

Figure 5. The Sparsely Populated Table.….………………………………………………………….7

# Introduction

When building a robotic device, often times, the algorithms that are applied are the easiest part to develop. The difficulties with robotics stem from the application of the algorithm in the real world. In the real world, sensors can fail, be rendered useless by inaccuracy, or even become less precise over time as the sensor ages. Real world implementation is an important part of computer science, and to gain familiarity with the real world, we implemented a robot using LeJos, a programming language that is used with the Lego Mindstorm robots.

# Background

LeJos is free software that adds on to the manufacturer software on a mindstorm “Brick”. Syntactically it is similar to Java, and functions as a useful go between from the click and drag software that is usually implemented and more complex coding actions used In an IDE such as eclipse. Eclipse is the IDE that we used to code the robot in thanks to its useful plugins that help with moving the software onto the robot.

The research that this paper focuses on is a seemingly simple task, find the darkest spot in the area to stay “hidden” from the light, which boils down to essentially building a cockroach.

3. Approach

**3.1 The Sensors**

**3.1.1 Ultrasonic Sensor**

The ultrasonic sensor served as the eyes of the robot. In reality, it sends out a “ping” and checks to see how quickly that ping returns. If readings are registered under a certain distance, the robot confirms that it “sees” an object and will carry out procedures to push it off of the table.

**3.1.2 Gyroscope**

The gyroscope tracks how far the robot has rotated. This was discovered late into the design process, but was included to try and increase the accuracy of the robot’s turns. For each turn, the gyroscope is reset, then queried to see how far it has turned from that reset position.

**3.1.3 Light Sensor**

**3.2 The Robot**

To properly map the movements made by a cockroach, we decided to use a slightly aggressive form of simulated annealing. Simulated annealing is the use of probability to discern what action should be taken. It is similar to most hill climbing algorithms, where the algorithm attempts to locate the best case or best “location” of the current states around the algorithm. For a robot, this means checking the surrounding area to see if the terrain around it was lighter or darker, and depending on that, make a decision on which direction to move in. The difference between hill climbing algorithms and simulated annealing algorithms are that simulated annealing includes the possibility of making a move that shifts the algorithm to a less than ideal state in the hopes that it will locate the global maximum, rather than the hill climber algorithm which will only make moves that move it to a better state, and is therefore more likely to find a local maxima. This difference is key, and implemented using probability, and a cooling rate. Simulated annealing uses a starting temperature, and a cooling rate to adjust how long the algorithm will run.

**3.3 Classes**

**3.1.1 Cockroach**

**3.2.2 Eyes**

Eyes.java is the main implementation for all the Ultra-sonic sensors. It simplifies the control for the ultrasonic sensors. It contains all the distance methods, which measure the distance between the sensor and the object being scanned.

This class exists merely to help keep the code readable, as it is much easier to call its only function getDistance() rather than repeatedly assign the variables within.

**3.2.3 Gyro**

Gyro.java is the implementation for all the gyroscopic sensor control. It simplifies controlling turning, and contains a method for getting the turning distance that the agent has turned.

This class exists merely to help keep the code readable, as it is much easier to call the function getDistance() rather than repeatedly assign the variables within. Gyro contains one extra reset() function, which allows it to be set back to zero before counting the degrees turned.

**3.4 Helper Functions**

**3.4.5 testEyes(eyes)**

testEyes() is a debugging method, used in calibrating the Ultrasonic sensor. It prints out the range that an object is detected at intervals so that an optimal range could be found.

**3.4.6 rotate(pilot, gyro, angle, eyes, touch)**

Rotate() is a combined function—initially, turning and scanning were separate actions, but it made more sense to simply scan every time the robot did any turn.

To do this, the robot turns slowly while the gyroscope tracks the difference in the starting and current angle, moving until either the ultrasonic sensor detects something within a certain scan range (.2 meters, though in practice this turned out to be slightly less than that due to the ultrasonic sensor’s variance) or the gyroscope detected that the robot had turned an amount equal to 5 degrees less than the desired angle. This 5 degree adjustment accounted for the gyroscope’s natural error as well as the stopping time required by the robot.

**3.5 Running the Program**

Summarize the steps taken in the program

**3.6 Testing the Program**

words

**Test Situation 1**

words

**Test Situation 2**

**words**

# **Results**

words

**4.1 What Went Wrong**

words

**4.2 What Would Be Changed**

words

# Conclusion

words

# Appendices