**Finding a Solution**

**Using Hill Climbing Strategies**

**CS491**

**Special Topics (AI)**

April 24, 2015

By Cody Kubicek

And Tyler Rusch

**Abstract:**

Contents

[1. Introduction 1](#_Toc412577616)

[2. Background 1](#_Toc412577617)

[3. Approach 6](#_Toc412577618)

[4. Results 9](#_Toc412577619)

[5. Conclusion 10](#_Toc412577620)

[6. Appendices 11](#_Toc412577621)

List of Figures

Figure 1. gsdfasdfasd……………………………………………………………………………………1

# 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 Mind-storm robots. LeJos is free software that adds on to the manufacturer software on a Mind-storm “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.

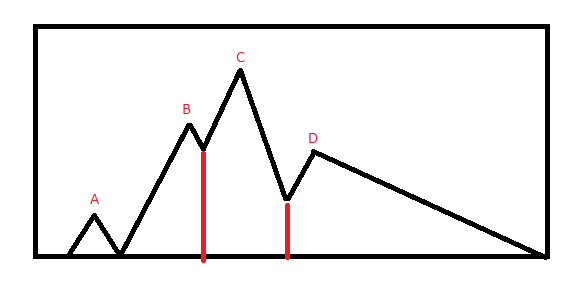
# Background

**2.1 Hill Climbing**

In order to search the environment, two local search strategies were employed: Steepest Ascent and Simulated Annealing.

**2.1.1 Steepest Ascent**

Steepest Ascent is the greedy application—check the possible moves and always take the best of them. When the available moves are all worse than the current position, the program terminates. Though this is fine for perfectly gradient climbs (there is one, even slope to the peak), it runs the risk of hitting local maximums and completing before a true maximum is found.

****

For instance, unless the starting position is somewhere between the two red lines, a steepest ascent approach will never find the true maximum for this scenario. It will look to the right of B/left of D and see that the new position would be worse, then retreat to the local maximum since it is unable to see the entire environment.

**2.1.2 Simulated Annealing**

One attempt to solve the local maximum problem is the Simulated Annealing approach.

* 1. **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 steepest ascent 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 steepest ascent 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 steepest ascent algorithm which will only make moves that move it to a better state, and is therefore more likely to find a local maxima. Simulated annealing is 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. This also allows for simple mathematics, as the temperature is adjusted once each run of the algorithm, using basic algebra we could easily approximate how many moves the robot would make, and how long the move would approximately take.

* + 1. **Detecting Boundaries**

words

**2.2.2 Detecting Light**

words

# Approach

**3.1 Classes**

**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.

**3.1.3 Light**

words

**3.2 Helper Functions**

**3.2.1 move(DifferentialPilot, Eyes, int)**

The move function takes over Pilot.travel() in order to allow the eyes to watch out for walls as it is moving. If an object comes within an estimated distance, the robot will stop completely to stay safe. It is worth noting again that the distance needs to be greater than expected due to the reasons explained in 2.2.1 Detecting Boundaries.

With simulated annealing, this actually has the effect of “eating” moves—the robot will choose a direction that would lead it into a wall, see it cannot move any closer to it, and the temperature would be cooled without really attempting a new location. This was determined to be a non-threatening issue, as the runtime is long enough that having enough repeated instances in order to affect the objective negatively has an incredibly low chance of happening.

**3.2.2 rotate(DifferentialPilot, Gyro, double, Eyes)**

Similar to move, rotate is brought to its own function (away from Pilot.rotate() )to add the functionalities of wall detection from the Eye class and added precision from the Gyro class. The only difference from move() aside from the motion is that instead of stopping entirely when an object is determined to be too close, the robot will adjust backwards and continue turning in order to keep true to right angles as much as is possible. Additionally, the scan range for determining if something is too close is much smaller (slightly less than half in the given code), as when rotating the eyes will be turned to face the wall instead of seeing them from an angle (see Figure x above for why this matters)

**3.2.3 checkProb(double, double, double)**

I will write this after Math.

**3.3 Running the Program**

words

**3.4 Testing the Program**

words

# Results

**4.1 Improving The Results**

# Conclusion

# Appendices

**6.1 Cockroach.java**

**6.2 Gyro.java**

**6.3 Eyes.java**

**6.4 Light.java**