TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Design of a maze solving robot using Lego MINDSTORMS

Citation for published version (APA):
van Putten, B. J. S. (2006). *Design of a maze solving robot using Lego MINDSTORMS*. (DCT rapporten; Vol. 2006.057). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/2006

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 13. Aug. 2022

Design of a maze solving robot using Lego MINDSTORMS

B.J.S. van Putten

DCT 2006.057

Attractive practical application of uncomplicated robotics

Bachelor final project

Supervisors

Prof. Dr. H. Nijmeijer
Dr. Ir. M.K. Camlibel

Eindhoven University of Technology
Department of mechanical engineering
Dynamics and Control group

Eindhoven, May 2006

Table of contents

1. Introduction

General introduction

In this report the capabilities and restrictions of the Lego® Mindstorms™ Robotics Invention System™ 2.0 combined with a programming code are explored. This is illustrated by designing, building and testing a maze solving robot. The case shows the attractive practical application of a relatively uncomplicated script and robot design. The Mindstorms package consists of a large number of Lego parts, just as every Lego set. In this package however a programmable brick is included, called the RCX. This brick enables the user to develop and run programs which control a certain built Lego creation. When replaced with a more sophisticated computer and a largely extended code, the technology described in this report can be used for an infinite number of applications, like automated transport systems and car parks. Much more obvious is the use of easy robotics in daily practice. Autonomous vacuum cleaners and lawn mowers are already available; they use relatively uncomplicated programs to make our daily life a little easier. In this report the concrete case of a maze solving robot is discussed.

Central goal and sub goals

The central goal of this report is formulated as follows.

Explore the capabilities and restrictions of the LEGO MINDSTORMS RCX 2.0 unit and LEGO hardware by developing a maze solving robot. The maze is set up by a black on white line pattern.

With respect to this central goal, a number of sub goals have been formulated to cover the entire process. The most important of those are

1. Design a hardware and software program for a line-following robot that has good properties in driving straight ahead and is able to detect crossings.

2. Expand the capabilities of this robot by adding the possibility to make choices on crossings and in doing so develop an easy maze solving algorithm.

3. Design a position recognition program so that the robot can even solve mazes containing loops and other hard structures.

4. Point out the restrictions of the Lego Mindstorms set and improve the results of the maze solver by means of improving both software and hardware.

Organisation of the report

First, chapter 1 describes the key attributes and a brief history of Lego Mindstorms. The unique feature of Lego Mindstorms, the programmable unit RCX, is explained in chapter 2. Chapter 3 deals with several of the programming codes available for the RCX and also a choice is made for this particular case. In chapter 4 the maze itself and the basics behind a maze solving algorithm are described, which leads to the design of an easy maze solving robot in chapter 5. Chapter 6 concerns improving the easy maze solver of chapter 5, so that it meets the boundaries set by the sub goals. In chapter 7 the building and testing results are presented. Chapter 8 concludes this report. In the conclusion a distinction is made between conclusions on Lego Mindstorms and the programming code in general and conclusions concerning the practical case of this maze solving robot.

2. Introduction in Lego Mindstorms

The development of intelligent and programmable Lego products has started in the early 1980's, when Lego established the Educational Products Department. This led to the release of the first computer controlled Lego products in 1986. Research had continued and computers were become smaller, as in 1998 Lego introduced the first Lego Mindstorms and Robotics Invention System. This kit included not only the usual Lego parts, but also a programmable brick, called the RCX. The main component of the RCX is a Hitachi H8 microcontroller. In 2000 Lego released the next series of the RCX, the 2.0. Its features are listed below, [1].

- 16 kb internal ROM
- 512 b internal SRAM (for firmware)
- 32 kb external SRAM (for programs)
- 16 MHz clockspeed
- 9V operating voltage (6 AA batteries)
- LCD display
- Infrared serial communication
- 3 input and 3 output ports
- size about 10x6x4 cm
- weight about 250 g

In the summer of 2006, Lego will introduce the latest and totally different version of the programmable brick, the NXT, [2].

2.1 The RCX

First, a closer look is taken on the structure of the RCX. The RCX is best described as having different layers of functionality. It is convenient to start with the bottom layer, the hardware and work upwards to the top layer, the user developed program.

Hardware layer

The hardware layer consists of all the hardware included in the RCX, the microprocessor, display, on board memory and other electronic parts. This is the basic layer which determines the boundaries of the RCX capabilities and which links the RCX to its external parts, such as sensors and actuators.

ROM layer

The ROM layer is the first software preprogrammed layer of the RCX. It links the processed user commands to the hardware of the system. The ROM layer also provides the opportunity of downloading a second piece of software to the RCX, called the firmware.

Firmware layer

The firmware interprets bytecodes translated by a compiler from the user program. The standard firmware is able to run programs written in one of the official languages, such as RCX Code and Robolab, but also those written in NQC. This firmware can be replaced when using other programming languages, in order to achieve better functionality.

Software layer

This layer is not actually present on the RCX, but written by the designer and compiled on a host computer. To achieve maximum functionality, many programming codes are available and usable on the RCX, when the right firmware is installed. The compiled code is transferred to the RCX by an infrared tower and the infrared port on the RCX.

As one has obtained insight in de structure of the RCX, a programming code can be chosen, chapter 3, and a design for a robot can be made as in chapter 5 and 6.

3. Programming codes

After the basic structure of the RCX is explored, one can take a closer look at the available programming codes. Some of these languages use the standard firmware included in the Lego set, others use different firmware. The most common languages are RCX Code, NQC, pbForth and LegOS. The key features of these languages are discussed briefly, for a more detailed introduction is referred to Extreme Mindstorms (2000). [3]

RCX Code

RCX Code is the language that comes with the Lego Mindstorms kit. The language is linked to a graphical interface, which enables easy basic programming. The code is easy to use, no programming experience is needed to design a program in RCX Code. This however also limits the possibilities and in larger programs the structure gets unclear. Its large advantage is the fact that it communicates with the infrared tower without trouble, no external interface or program is necessary as in most other languages.

NQC

Not Quite C, NQC, is a language developed by Dave Baum, especially for the RCX. As its name implies, it is based on the programming code C, but adjusted to fit the RCX. NQC uses the standard firmware, which leaves about 6 kB of free space for the user defined programs. It may not seem much, but is sufficient for most programming applications. This language is relatively easy to use and there is enough information available concerning NQC, for example a tutorial by Mark Overmars (1999), [4]. For communicating with the RCX and compiling the programs, an external interface is needed.

pbForth

pbForth is a language based on the higher programming code Forth, which is developed by Charles Moore about thirty years ago. This makes the language extra interesting, because at that time, computers had capabilities similar to the RCX. It is a low-level and interactive language specifically designed to be used on computers with few resources. pbForth can cope better with many variables than the previous two languages, so more complicated codes can be written than in RCX Code or NQC. pbForth requires an interface and new firmware for the RCX.

LegOS

LegOS, like NQC, is based on the language C. It has been created by Markus L. Noga and has been used and refined for about thirty years. LegOS is, even better than pbForth, able to deal with variables, because C provides an unlimited number of variables. In addition, legOS provides a number of library functions that give the programmer a level of control no other language can match. A disadvantage of this language is that it is more complicated than the previous three, because the essentials of C have to be learned. Besides, an interface still is needed and the RCX is run on different firmware.

Because of the disadvantages of the more difficult languages like pbForth and legOS and the short time span, in this project the choice has been made to work with NQC.

This language has enough functionality, is relatively easy to learn and use and makes use of the standard firmware.

It is possible to write NQC codes in a program like Notepad, but the problem is then how to compile the code and download it to the RCX. Therefore, the interface made by Mark Overmars, Bricx Command Center is very useful. This program is refined later on and is still available on the internet, [5]. It has some useful features, like direct control of the RCX, but its key feature used in this project is in compiling and downloading the program to the RCX.

After the choice has been made for the programming code, a closer look can be taken at the maze that is to be solved, chapter 4, and the design of a maze solver, chapter 5.

4. Maze and maze solving

This report deals with the concrete case of designing building a maze solving robot. Now a look is taken at the maze itself and which restrictions are needed.

To be able to design a maze solving robot, the boundaries need to be defined by the maze itself. When attempting to solve a maze, one should take a close look at the structure on which mazes are built. By far most mazes are drawn in the same way, often only their size is adjusted to change the level of difficulty. Solving these mazes however is rather easy, when one is familiar with this level of mazes. By consequently following one of the walls, either the right or the left one, solving any maze is just a matter of time. Figure 4.1 shows a maze that looks rather easy, but it shows the essentials of the maze solving algorithm needed later on. The maze itself can get much larger and include crossings and loops, but the way it can be solved is the same for most mazes.

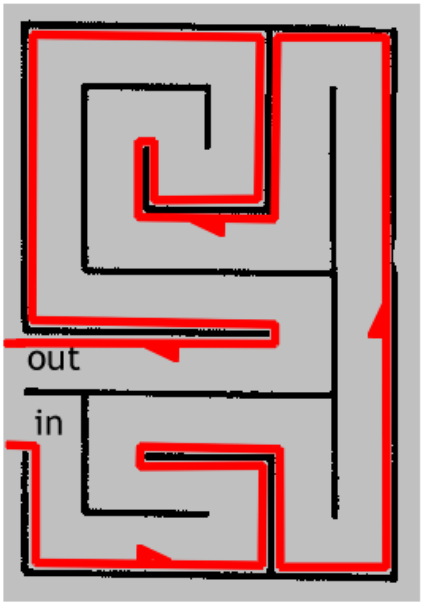

Figure 4.1 Easy maze and maze solving

For a start, the maze is set up by a wide black line on a large sheet of white paper. The robot should be able to trace the either left or right side of the track and so find the exit. This maze can be adjusted to suit the needs and to expand the challenge, but the basics always are a black line on a white floor.

After the boundaries for the maze solver are set, the design can be started. In chapter 5, a possible hardware and software setup for the easy maze solver is discussed. This robot is able to solve any maze, but does not yet meet all the requirements defined in the goals. Therefore, the robot needs major adjustments in both hardware and software. Finally, an intelligent robot that is able to find its way through a maze without detour after exploring it once is discussed in chapter 6.

5. Easy maze solver

After the maze is determined, a maze solving robot can be designed. The design of both hardware and software is presented in this chapter.

5.1 Optimum shape

The size of the maze and the number and size of parts in the Lego Mindstorms system decide the general size of the robot. It can not be larger than about 20x20x20 centimeters.

Because it will have to be able to steer, there are several options. The two most important of those are a fully rotational robot and a robot with a rack and pinion system as used in cars. These two options are discussed briefly and based on the advantages and disadvantages, a choice is made.

Fully rotational robot

A fully rotational robot makes use of two motors, each driving one side of the robot. A disadvantage of this type of drive is the fact that it hardly ever drives perfectly straight ahead. The motors are normally not totally identical and just some difference in play on the gear set causes different rotational speed of the wheels, leading to a steering action. The drive itself can be achieved by wheels or by caterpillar links. When using wheels, the best option is using only one pair of wheels, but this would cause problems in stability. This is easily solved by adding a pivoting wheel to the front or the rear of the robot, but this will add some turning circle. Because of the Lego system, the pivoting wheel can not be correctly aligned with its axis, which adds some unexpected behavior when turning. Although the resistance by friction of this option is larger, the caterpillar links are the best solution for a fully rotational robot, due to their predictability compared to the use of wheels and a pivoting wheel.

Rack and pinion robot

The rack and pinion steered robot also makes use of two motors, only in this case, one drives the robot and the other is used for steering. This works in principle like a normal passenger car. The large disadvantage of such a steering device is the large turning circle compared to the fully rotational robot. An advantage however is the better driving straight ahead capability, which can lead to a higher speed through the maze.

Taking the advantages and disadvantages of both systems in account, the best option is the fully rotational robot, because of its turning capabilities, which are absolutely necessary in navigating through the maze.

Sensors

For being able to drive around through the maze, the robot needs a way to detect the black on white line pattern. The Lego Mindstorms system includes a light sensor for this purpose. The light sensor is not just a sensor, but also has a light emitting diode on board. This makes it an absolute as well as a reflection sensor. It can read a bright light that is pointed towards the sensor, but also detect different levels of absorption

of the emitted red light. This last capability enables it to distinguish white paper from black and even some shades of gray and is used to follow the maze.

The hardware needed for a robot as desired here is listed below.

- RCX 2.0
- Two motors
- A number of light sensors (maximum of three)
- Caterpillar links
- A Lego chassis structure to mount all parts

For the easy maze solver, one light sensor is sufficient, which is available in every Lego Mindstorms kit. With a correct algorithm programmed in the RCX, as is discussed in the next subsection, it is able to follow a black line and solve mazes. The position of this sensor has to be on a relatively large distance from the turning axis. By doing so, it is able to detect small changes in direction and react swiftly.

5.2 Line following algortihm

The line following problem is not really hard to tackle. The real challenge however is to optimize the line following capabilities. In designing a line follower, the easiest solution is found in building an algorithm that changes the direction of the vehicle when it is either on the black line or on the white paper. In this program lines as follows are used.

```
If (on black line)
{
        Forward (left motor);
        Off (right motor);
}

Else (on white)
{
        Forward (right motor);
        Off (left motor);
}
```

The entire program is included in Appendix A.

By doing so, the robot zigzags on the line, but because the motors are not reversed when zigzagging, it makes progress. The path it follows is illustrated in figure 5.1.

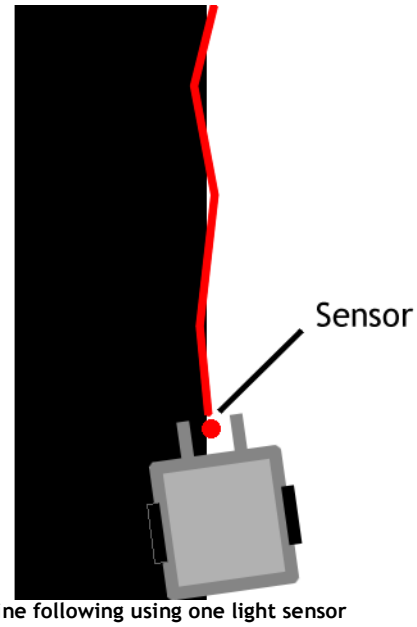

Figure 5.1 Line following using one light sensor

Of course, this is not the optimal behavior of the robot, because one would like it to drive straight ahead when following a straight line. This program is unable to accomplish that goal, but it has some opportunities in solving other problems. It easily follows a curve which radius is larger than the width of the robot. Another advantage is the fact that it always chooses one side of the line to follow. This technique is the easiest way to solve a maze, as described in chapter 4.

A robot that is built according to the guidelines discussed above, is able to solve a maze of any difficulty. The way it solves the maze is not the most sophisticated, but because a maze is considered to be randomly shaped, it is inevitable. There are some limitations to the behavior of the robot.

- The robot is not able to drive straight ahead, as required in one of the sub goals
- The robot is only able to navigate through mazes with tracks about twice as wide as the robot itself
- The robot does not use the information that it receives by solving a maze, like being able to find the exit much faster after exploring it once

The next section deals with solving these problems and increasing the overall performance of the maze solving robot.

5.3 Driving straight ahead when possible

The basic structure of the robot described in the previous section is used to expand its capabilities. For driving straight ahead, one light sensor is not sufficient, the robot always zigzags over the black line. There are several options to make the robot follow a straight or curved line on the white paper floor. One of those is adding an additional light sensor. A second light sensor is not included in the Lego Mindstorms kit. The problems of the single light sensor however are solved by adding this second sensor. When one light sensor is placed at the right side and one at the left side of

the black line, the robot is able to drive straight ahead when it knows the line is between its sensors. Also curves are tackled somewhat more precise.

The algorithm has to be adapted to deal with the new circumstances. In the case of only two light sensors, the robot will steer to the left when reading a black line on the right and shortly turn right when receiving a black line on the left. When both sensors read the white paper floor, the robot drives straight ahead. The algorithm will include the next essential lines, a complete code is included in Appendix B.

```
If (sensor_right on black line and sensor_left on white)
{
        Forward (left motor);
        Off (right motor);
        Wait (time);
}

Else if (sensor_right on white and sensor_left on black line)
{
        Forward (right motor);
        Off (left motor);
        Wait (time);
}

Else (sensor_right on white and sensor_left on white)
{
        Forward (left motor + right motor);
}
```

The path that the robot follows using this set up is shown in figure 5.2.

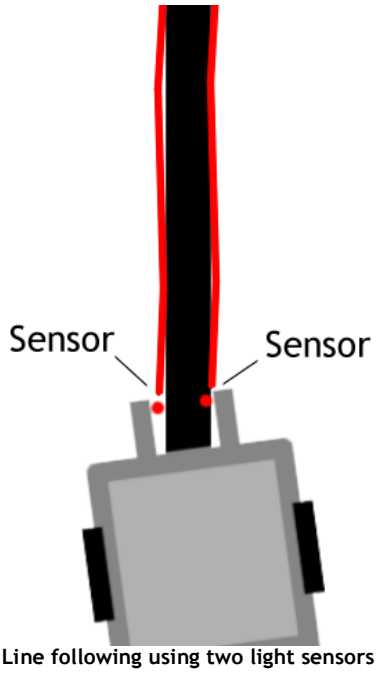

Figure 5.2 Line following using two light sensors

The complete code of Appendix B. includes the possibility of a crossing, on which the robot in this case turns around and checks the number of options. In the case described above, the accuracy and speed are determined by the distance between the sensors, but also by the turning time when reading the black line. A low value for this parameter compared to a higher value results in a lower speed, but higher accuracy.

A side effect of the fact that the robot now uses two light sensors each following one side of the track is that either the track has to be narrowed or the robot grows much wider. Because of reasons concerning the application in this case and an eventual link to practical use of a such design in for example an automated parking system, as mentioned briefly in the introduction, the choice has been made to narrow the track. This causes some problems, for example the robot can no longer follow the line in a 90 degrees turn, because its turning circle is too large.

Another side effect is that the essential maze solving option is lost. As the robot no longer automatically follows one side of the track, this is added manually. For that matter the robot has to be able to detect a crossing, so that it can turn right when possible. This can be done by adding a subroutine when both sensors detect a black line, like in the complete code of Appendix B. This however is not the most reliable and sophisticated solution.

As the RCX has room for 3 sensors, a third light sensor is added. This third light sensor is placed centrally on the front of the robot. The configuration is shown in figure 5.3.

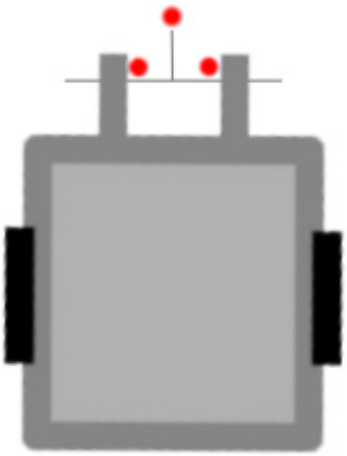

Figure 5.3 Robot with 3 sensor configuration

The use of this third light sensor and the design of a more sophisticated maze solver are explained in chapter 6.

6. Design of a more sophisticated maze solver

Using the hardware and software structure as described in the previous chapters, a more sophisticated maze solver can be built. This robot first explores the maze using the strategy discussed in chapter 4. In the second run, it uses the gathered information of the first run to find the exit without detour. It 'learns' the maze. The hardware of chapter 5 is used and the code is expanded to be able to store the choices the robot has made.

6.1 Basics of an intelligent maze solving robot

The maze is adjusted to make the robot able to distinguish the one crossing from the other. This is done by assigning a shade of gray to each crossing, instead of the black and white used for the line and the floor. The light sensors can detect the percentage of reflection of each crossing. Of course, every crossing has to have a unique shade of gray, so the robot can see the difference.

For recognizing the crossing, the robot uses a third light sensor. This sensor is placed in the center and normally follows the black line. When it detects a crossing, a subroutine is started in which the robot first tries to turn right, according to the first easy maze solver. When turning right is not possible, it drives straight ahead. A variable is assigned to each shade of gray, so that it is unique for each crossing. This variable is used to store the solving option at each crossing. The essentials of this algorithm are explained below according to figure 6.1.

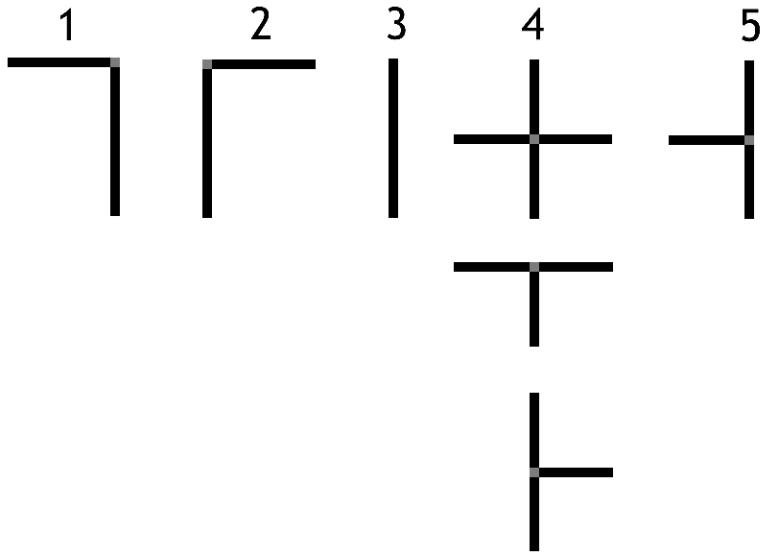

Figure 6.1 Possible crossings

1 A 90 degrees turn to the left. The left light sensor detects a crossroad, while the center light sensor detects the white paper floor. The robot drives to the center and turns left. There is no variable needed and the crossing does not have to be marked, because the robot always turns left here.

2 A 90 degrees turn to the right. Instead of the left sensor, this time the right light sensor detects a crossroad, while the center sensor detects the white floor. The robot drives to the center and turns right.

3 A dead end. All three sensors will detect the white paper floor, so the robot makes a 180 degrees turn and returns to the previous crossing, where it continues its search for the exit.

4 These three crossings have one thing in common, there always is the possibility to turn right. The robot detects the shade of gray of the crossing and turns in order to check for the option of a right turn. This option is available, so the robot adds 1 to the crossing assigned variable when turning right.

5 No possibility to turn right. The robot detects the shade of gray of the crossing and turns in order to check for the option of a right turn. Because this option is not available, the robot turns back and continues its way, straight ahead. This adds 2 to the variable.

The turns are determined by a time parameter. Because every turn is 90 or 180 degrees, two parameters do to determine the turning on the entire map of the maze.

6.2 Theoretical cases

Because the robot uses variables to store its choices on the several crossings, let us now consider some cases to check whether this technique really works or not.

Case 1

The maze in this case is a crossing of 2 roads, where the exit is to the left, though all other options are dead ends. See figure 6.2.

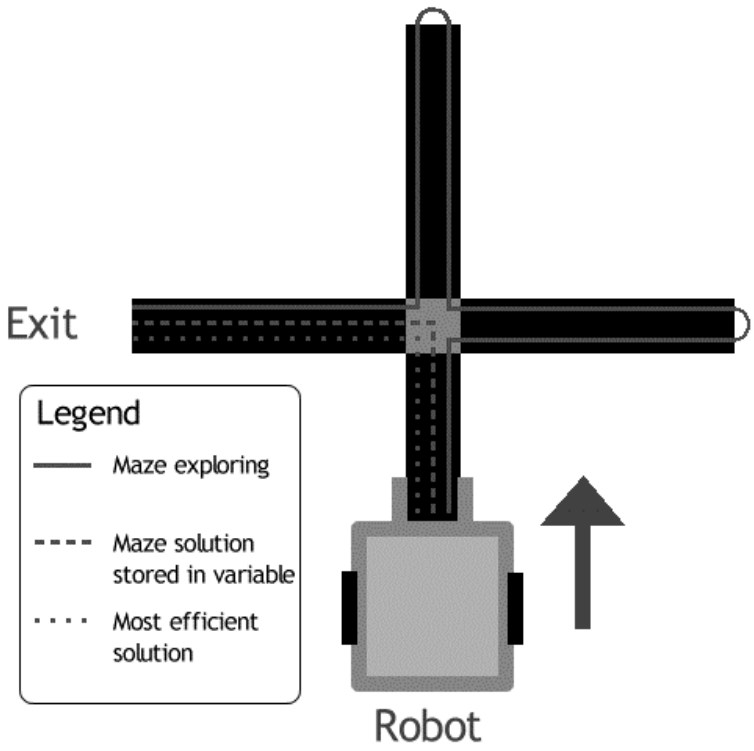

Figure 6.2 Case 1

The robot approaches a crossing where it can turn right. The central sensor detects the shade of gray, after turning right, the variable is set from 0 to 1. Because all

options are a dead end it returns 3 times to the crossing, so the final value of the variable is 3. When the robot now approaches this crossing again, it has got the variable 3 stored which corresponds with a turn to the left. So now the robot will go to the left without trying out all the other options and find the exit as fast as possible. This solution is in this case the most efficient solution.

Case 2

The piece of the maze the robot encounters in this case is a T-crossing, the exit again is to the left. See figure 6.3.

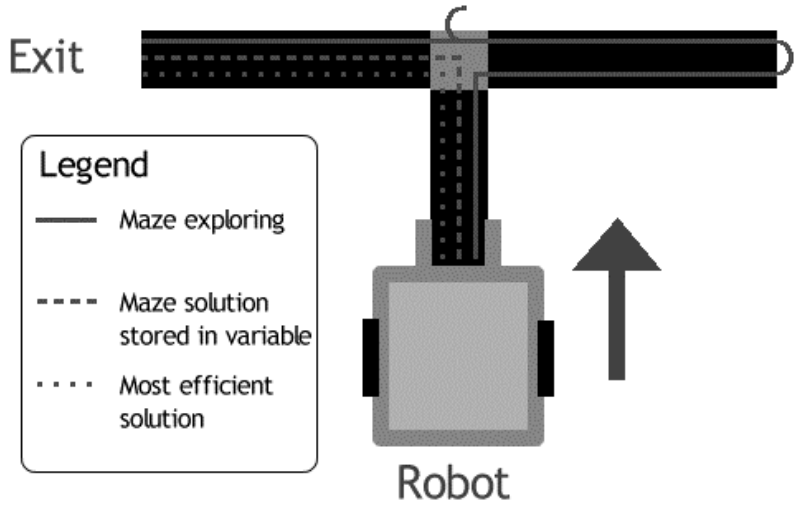

Figure 6.3 Case 2

In this case, the robot approaches a crossing which has another shade of gray than in the previous case. Because it is able to detect the difference, it uses another variable. First, the robot tries to turn right. Because this is possible, it stores 1 in the variable. It reaches a dead end and turns back to the crossing, where it now is not able to turn right. It drives straight ahead and adds 2 to the variable, so again it is set to 3 as the robot reaches the exit. After reaching the exit and the robot is set back to the starting position, it will automatically turn left on the crossing, which is most efficient.

Case 3

The robot reaches a loop in the maze. See figure 6.3.

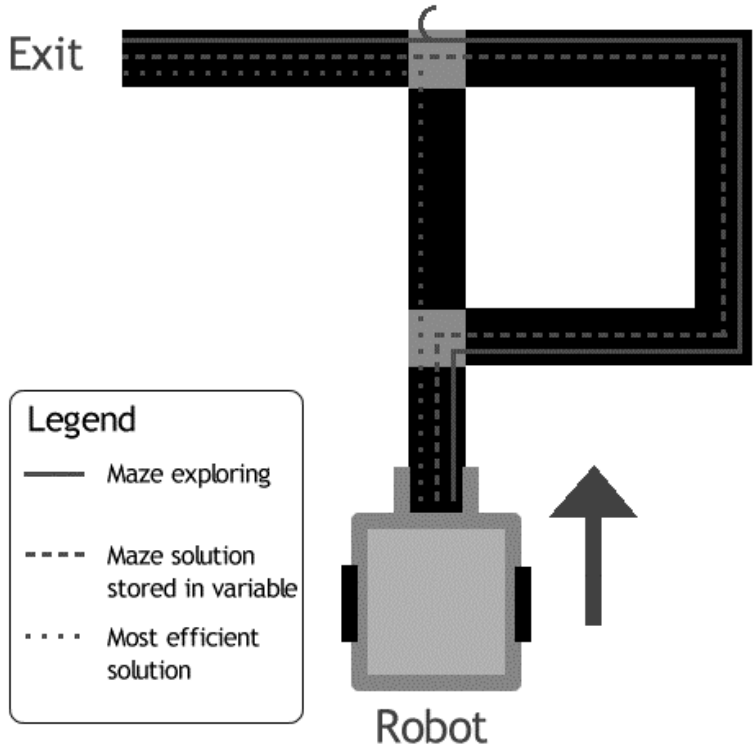

Figure 6.3 Case 3

Because the robot always turns right, this loop is not a problem. At the first crossing it turns to the right and adds 1 to the variable corresponding to this crossing. This leads to the next crossing, where it has to drive straight ahead and adds 2 to the corresponding variable. Then it finds the exit. As is clear, this is not the most efficient way to find the exit. With this program and hardware however, this problem is not solvable and some further research can be done to solve it.

Case 4

The robot approaches a loop in the maze, which connects two of the possible options at a crossing. See figure 6.4.

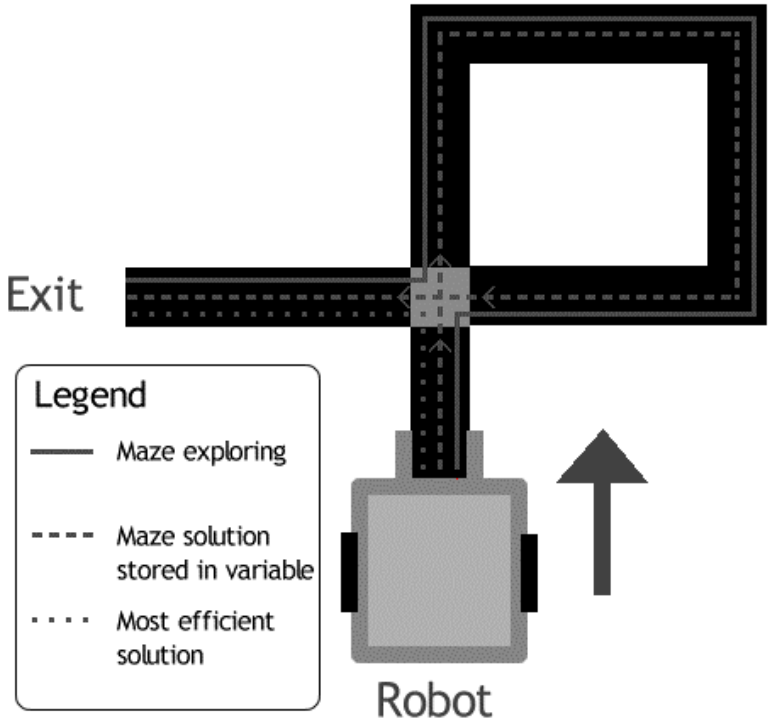

Figure 6.4 Case 4

The problem in this case and many similar cases is in saving the correct value for the crossing parameter. When the robot reaches the crossing, it turns right. 1 is added to the variable. This road however leads back to the same crossing. If it led back to the same side of the crossing as where it started, this would have caused no problem. In this case, it leads to another option at the crossing, whereas the robot gets confused. Namely, it adds another 1 to the variable, because of turning right again, and finds the exit. The variable is now set to 2, though only 3 would lead to the most efficient solution. With 2 stored, the robot also finds the exit, just not as efficient as should be, as seen in figure 6.4.

6.3 Orientation of the robot

A solution for this problem can be found in storing the orientation of the robot in a variable too. Important in this case is the knowledge that the maze only contains 90 degrees turns and dead ends, so that 4 values are sufficient. The following agreement about the variable storing the orientation of the robot is used. See figure 6.5.

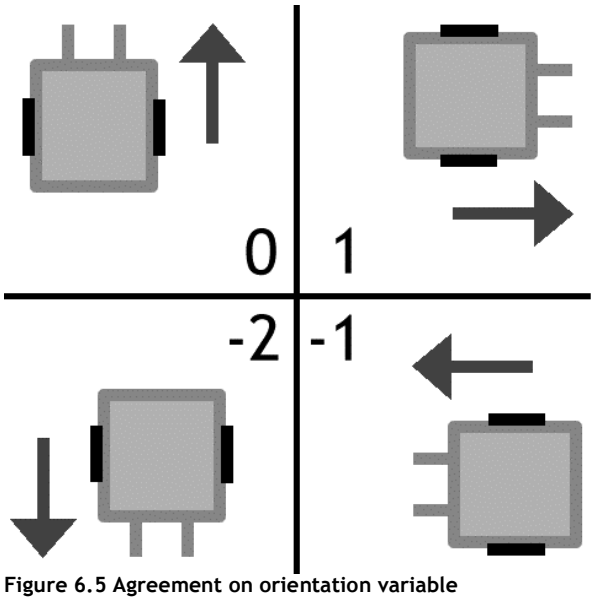

Figure 6.5 Agreement on orientation variable

The orientation variable is increased by 1 when the robot makes a 90 degrees turn to the right, while it is decreased by 1 when turning left. A 180 degrees turn decreases the variable by 2. When the variable reaches 2, it will be changed in -2, while -3 is changed in 1. This ensures the use of 4 values in top view, up is equal to 0, right corresponds to 1, left to -1 and down to -2.

The orientation of the robot can now be used to solve the problem as described above. After turning right, the orientation variable is set to 1. When the robot returns to this crossing again, it is expected to approach from the side it has left. This means, in terms of the agreement about the orientation variable, that its expected orientation is -1 when returning to that crossing. In the case above, the robot has orientation -2 when returning to the crossing. By adding the absolute value of the difference between the real and the expected value of the orientation variable, the crossing variable is corrected. In case 4, the variable is corrected to 3, which leads to the most efficient solution in the second run.

7. Building and testing the robot

The robot is built to the specifications given in the previous two chapters. This chapter is about testing its abilities in line following and maze solving. First the basic functions are tested and discussed, next some cases are executed to check whether the robot is able to tackle some problems. Finally the robot is tested on a real maze, including two crossings, dead ends and some 90 degrees turns. The problems encountered in testing the robot are discussed in their respective subsections.

7.1 Building

In building, no real problems occurred. The use of the standard Lego parts limits precise prototyping, but in this case the maze is also built to the specifications of the robot. This means the width of the track is adjusted to the distance between the two outer light sensors to solve this problem in accuracy.

7.2 Basic functions

The functions of the maze solver are tested on a test track. A map of this track is included in Appendix D. Test track. All tests are run 10 times and the results are presented in a percentage scale. When problems are encountered, a sketch of the situation is added.

- moving forward and backwards 100%
- steering 100%
- sensors (both percentage and edge-mode) 100%

7.3 Cases

Detection by sensors

- crossing (different shades of gray) 100%
- dead end 50%

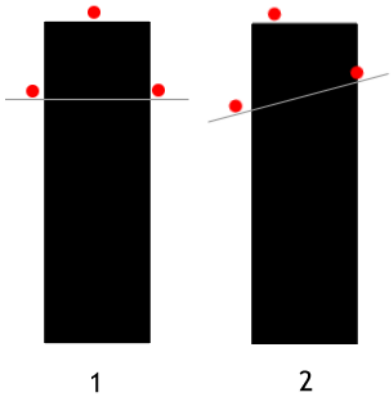

Figure 7.1 Dead end

Problems occur when the robot is not totally aligned with the track, as shown in figure 7.1, situation 2. The three light sensors not all detect the white paper floor as

should be in situation 1, one of the two side sensors detects the black track. This causes the robot to see the dead end as a 90 degrees turn; it turns 90 degrees and loses the track. This problem is likely to occur often, because the light sensors even do not allow a small angle between the robot and the track, which is needed for other applications.

- 90 degrees turn 70%

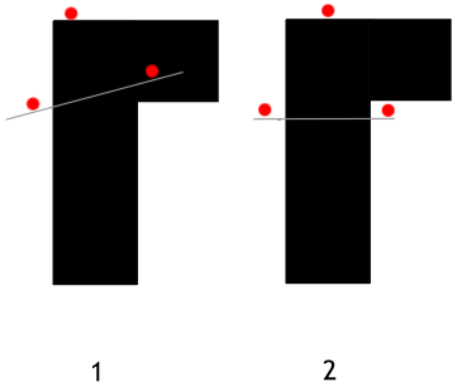

Figure 7.2 90 degrees turn

In this case, see figure 7.2, problems occur when the robot is perfectly aligned with the track. It detects the 90 degrees turn as it were a dead end, because all three sensors are on the white paper. Because the robot more often is not perfectly aligned, this problem occurs not as often as the previous one.

- detect end (shade of gray) 100%

Action by motors

- crossing (slow down and drive to center) 100%
- dead end (drive to center and turn 180 degrees) 60%
- 90 degrees turn (drive to center and turn 90 degrees) 70%

For the 60% and 70% scores, the inequality between the both sides of the robot is to blame. Because of the use of the standard Lego parts and the Lego building system, the resistance from the motors to the tires is not equal for both sides. Additionally, the slip between the tire and the track is not entirely the same every time the robot makes a turn.

While testing a problem occurred concerning the battery level of the RCX unit. The turns are all determined by time parameters, so as the battery level drops, the robot covers less distance and turns are not completed. To test the behavior of the battery pack, the battery level is measured before and after an hour of testing. It turned out that the voltage had dropped from 9,08 V to 8,12 V. This loss of power makes the robot unable to solve a maze with a very long route to the exit and causes the programmer to check and fine tune the time parameters every time the robot is run.

7.4 Maze solving

- Solve the test track from start to exit 20%
- Use the information to solve the second time without detour 20%

In solving the test track from start to exit, all problems encountered in previous sub cases cumulatively lead to the low scores. In 2 of the 10 cases however, it solves the maze. The information gathered while solving, led in 100% of these cases to solving the maze the second time without detour. Of course, to draw conclusions about the 'learning' abilities of the robot based on two measurements is not reliable, so another test is performed.

- Find the exit in a maze with just one crossing 70%
- Use the information to find the exit at once 70%

Simplifying the maze ensured a much higher score on finding the exit. It also shows that the robot still makes use of the gathered information and finds the exit the second time without detour every time it solves the maze.

8. Conclusions and recommendations

Research has been done on the hardware and the software for a maze solving robot. It has included both theoretical and practical research, the conclusions are presented in this chapter. First, some conclusions are drawn on the Lego Mindstorms kit and the programming code NQC in general. Next the conclusions on the maze solving robot are presented. The report is concluded by recommendations on both the hardware and software in general and the practical case of the maze solving robot.

8.1 Conclusions on the Lego Mindstorms kit and NQC

Lego Mindstorms has proven to be a good basis for an intelligent robot, which is able to 'learn' with a rather small program. The main strength of the Lego kit lies in easy prototyping, whereas the programmable brick RCX offers some great features in very to the point programming. In this case, the technology is used to design and build a maze solving robot, which 'learns' the maze. Of course, this practical application is only exemplary for the use of Lego Mindstorms and the programming language NQC. One can think of an infinite number of other practical applications. The capabilities of such a design however are limited by both the Lego kit and the programming code.

- The battery capacity of the RCX unit is actually too small for application in which parameters are used that are influenced by the battery level.

- It is impossible to build a robot with 100% reproducibility of results. Because of the easy prototyping and building using the standard Lego parts, the play of the mainly rotating parts is never the same. This causes problems when very accurate positioning is needed and it has effects on the reproducibility of the results of the robot.

- The programming code NQC itself also has some limitations, of which the use of maximally 32 variables is the most important. In more sophisticated applications, one will soon experience the lack of storage space for variables crucial to solve problems.

Summarizing, Lego Mindstorms and the language NQC provide a basis on which one is able to design and build robots with potentially very interesting technology, like the ability to 'learn' its environment. It has its disadvantages, but considering the relative ease of use the entire package offers a very useful tool to show sophisticated behavior of a modern robot.

8.2 Conclusions on the design of the maze solver

The maze solver turns out to meet the specifications set by the sub goals. It is able to solve a maze of which it has no more information than that the track is in black and the crossings and exit are marked by a typical shade of gray. In solving the maze using the technique of always following the right side of the track, it stores the essential information to find the exit the second time without detour. The design has its drawbacks, but it fulfills all the goals set in advance of this project. The resulting problems are listed below.

- The robot is unable to 100% accurate detect dead ends and 90 degrees turns. The problem with dead ends occurs when the robot is not entirely lined up with the track, but just in a steering move. In this case, it loses the track. Detection of 90 degrees turns is not reliable when the robot is exactly in line with the track. It then detects a 90 degrees turn as a dead end.

Of course, the robot encounters the general problems of the Lego Mindstorms kit and the language NQC.

- The battery capacity influences the power supplied to the motors. In the designed robot 90 and 180 degrees rotation and driving to the center of crossings, are determined by time parameters. As the battery level drops, the robot does not reach the expected position any more in the determined time. This causes the robot to lose the track or to miss possible options on crossings.

- The problem of reproducibility is showed clearly when running the maze solving robot several times over the test track.

- The limitations of the programming code NQC are not yet reached using the code for this maze solver and attempting to solve the test track. When trying to solve a real, much larger maze, the robot can run out of memory for its variables, because 2 are necessary for each crossing and at least 2 in general. In this case also the capabilities of the light sensor are reached, because the distinction between the shades of gray becomes smaller when more crossings are included in the maze.

8.3 Recommendations

The research presented in this report leads to several recommendations. Those are presented again divided in recommendations related to the hardware and programming code in general and recommendations for further research concerning the maze solving robot.

Hardware and programming code

- Find a solution for the rapid decrease of the battery level of the RCX. A possible solution is an external power supply, but this limits the maneuverability of the robot.

- When the practical application requires the use of many variables or harder programming structures, NQC will not be sufficient. Languages like pbForth and LegOS are more likely to be able to cope with more difficult problems.

Maze solving robot

- Entirely implement the structure of the orientation of the robot of chapter 6.3 in the maze solving code of Appendix C. The basic structure has been laid out, but it is not finished.
- Reconsider the most sophisticated code, see Appendix C., to make it able to find the exit of every maze without detour. For example, a solution has to be found for the problem of chapter 6.2, case 3.

- The use of time parameters for turning on crossings and turns is not highly sophisticated. A better solution can be found in the use of the light sensors to determine the turning time.

- When 100% accuracy and reproducibility of results is required, the design of the robot is better made from scratch than using Lego parts. Lego provides easy prototyping, but at the cost of precision.

Bibliograpy

[1] http://www.robotadvice.com/lego-mindstorms_robot.html (23-04-2006)

[2] http://www.lego.com/eng/info/default.asp?page=pressdetail&contentid=17278&countrycode=2057&yearcode=&archive=false (18-04-2006)

[3] Dave Baum, Michael Gasperi, Ralph Hempel, Luis Villa, "Extreme Mindstorms", Apress, Berkeley (CA), 2000

[4] http://www.cs.uu.nl/people/markov/lego/tutorial.pdf (28-04-2006)

[5] http://bricxcc.sourceforge.net/ (03-05-2006)

Appendix A. Line following code

```
#define LIGHT SENSOR_2
#define LEFT  OUT_A
#define RIGHT OUT_B

task main()
{
      SetSensor(LIGHT, SENSOR_LIGHT);
      SetPower(LEFT+RIGHT, 6);

      while (true)
      {
            if (LIGHT <= 45)
            {
                  OnFwd(LEFT);
                  Off(RIGHT);             //steer right
            }

            else
            {
                  OnFwd(RIGHT);
                  Off(LEFT);              //steer left
                  ClearSensor(LIGHT);
            }
      }
}
```

Appendix B. Line following code 2 sensors

```
#define LIGHTLEFT SENSOR_1
#define LIGHTRIGHT SENSOR_3
#define LEFT OUT_A
#define RIGHT OUT_B

task main()
{
      SetSensor(LIGHTLEFT, SENSOR_LIGHT);
      SetSensor(LIGHTRIGHT, SENSOR_LIGHT);
      SetPower(LEFT+RIGHT, 8);

      while (true)
      {
            SetSensorMode(LIGHTLEFT, SENSOR_MODE_PERCENT);

            if(LIGHTLEFT <= 45 && LIGHTRIGHT <= 45)   //crossing
            {
                  SetPower(LEFT+RIGHT, 1);
                  Off(LEFT+RIGHT);
                  OnFwd(LEFT+RIGHT);
                  Wait(200);                    //drive to center
                  OnRev(RIGHT);
                  OnFwd(LEFT);                  //turn
                  SetSensorMode(LIGHTLEFT, SENSOR_MODE_EDGE +7);
                  ClearTimer(0);

                  while (Timer(0) < 100)
                  {
                        if (LIGHTLEFT == 1)     //1 crossroad
                        {
                              PlayTone(440, 5);
                        }

                        else if (LIGHTLEFT == 2)      //2 "
                        {
                              PlayTone(660, 5);
                        }

                        else if (LIGHTLEFT == 3)      //3 "
                        {
                              PlayTone(880, 5);
                        }

                        else if (LIGHTLEFT == 4)      //4 "
                        {
                              PlayTone(1080, 5);
                        }
                  }
                  Off(LEFT+RIGHT);
                  Wait(100);
            }
```

```
            else if(LIGHTLEFT <= 45 && LIGHTRIGHT >= 45)
            {
                  SetPower(LEFT+RIGHT, 8);
                  OnFwd(RIGHT);
                  Float(LEFT);                  //steer left
                  Wait(15);
            }

            else if(LIGHTLEFT >= 45 && LIGHTRIGHT <= 45)
            {
                  SetPower(LEFT+RIGHT, 8);
                  OnFwd(LEFT);
                  Float(RIGHT);                 //steer right
                  Wait(15);
            }

            else
            {
                  SetPower(LEFT+RIGHT, 8);
                  OnFwd(LEFT+RIGHT);            //straight ahead
                  Wait(15);
            }
      }
}
```

Appendix C. Maze solving code

```
#define LIGHTLEFT SENSOR_1
#define LIGHTCENTER SENSOR_2
#define LIGHTRIGHT SENSOR_3
#define LEFT OUT_A
#define RIGHT OUT_B

#define TURN_1 140
#define CROSSING_CENTER 180
#define TURN_2 215
#define TURN_CENTER 125
#define TURN_3 110

#define MOTORPOWER_1 4
#define MOTORPOWER_2 2
#define MOTORPOWER_3 1

int a;       // variable crossing one
int b;       // variable crossing two
int end;     // variable for end recognition
int r;       // variable for global orientation of the robot
int r1;      // variable for orientation of the robot at crossing one
int r2;      // variable for orientation of the robot at crossing two

task main()
{
  SetSensor(LIGHTLEFT, SENSOR_LIGHT);
  SetSensor(LIGHTCENTER, SENSOR_LIGHT);
  SetSensor(LIGHTRIGHT, SENSOR_LIGHT);
  SetPower(LEFT+RIGHT, 3);

  a = 0;
  b = 0;
  end = 0;
  r = 0;
  r1 = 0;
  r2 = 0;

  while (true)
  {

    if(end == 0)
    {

      if (r == 2)
      {
        r = -2;
      }

      if(r == -3)
      {
        r = 1;
      }

      SetSensorMode(LIGHTRIGHT, SENSOR_MODE_PERCENT);
      ClearTimer(1);

      if (LIGHTCENTER >= 41 && LIGHTCENTER <= 42)
      {
        SetPower(LEFT+RIGHT, MOTORPOWER_2);
```

```
  Off(LEFT+RIGHT);
  OnFwd(LEFT+RIGHT);
  Wait(CROSSING_CENTER);
  OnRev(RIGHT);
  OnFwd(LEFT);
  SetSensorMode(LIGHTRIGHT, SENSOR_MODE_EDGE +8);
  ClearTimer(0);

    while (Timer(0) < 15)
    {

      if (LIGHTRIGHT == 1)
      {
        PlayTone(440, 5);
      }
    }

    Off(LEFT+RIGHT);
    Wait(50);

    if (LIGHTRIGHT == 0)
    {
      OnRev(LEFT);
      OnFwd(RIGHT);
      Wait(TURN_1);
      a += 2;
    }

    else if (LIGHTRIGHT >= 1)
    {
      Wait(50);
      a += 1; r += 1;
    }
}

else if(LIGHTCENTER >= 35 && LIGHTCENTER <= 36)
{
  Off(LEFT+RIGHT);
  PlayTone(220, 5);
  Wait(1000);
  end = 1;
}

else if(LIGHTCENTER >= 46 && LIGHTCENTER <= 47)
{
  SetPower(LEFT+RIGHT, MOTORPOWER_2);
  Off(LEFT+RIGHT);
  OnFwd(LEFT+RIGHT);
  Wait(CROSSING_CENTER);
  OnRev(RIGHT);
  OnFwd(LEFT);
  SetSensorMode(LIGHTRIGHT, SENSOR_MODE_EDGE +8);
  ClearTimer(0);

    while (Timer(0) < 15)
    {
      if (LIGHTRIGHT == 1) //
      {
        PlayTone(440, 5);
      }
    }

    Off(LEFT+RIGHT);
```

```
    Wait(50);

    if (LIGHTRIGHT == 0)
    {
      OnRev(LEFT);
      OnFwd(RIGHT);
      Wait(TURN_1);
      b += 2;
    }

    else if (LIGHTRIGHT >= 1)
    {
      Wait(50);
      b += 1; r += 1;
    }
}

else if(LIGHTLEFT >= 51 && LIGHTRIGHT <= 51 && LIGHTCENTER >= 51)
{
  SetPower(LEFT+RIGHT, MOTORPOWER_1);
  Off(LEFT+RIGHT);
  OnFwd(LEFT+RIGHT);
  Wait(TURN_CENTER);
  OnRev(RIGHT);
  OnFwd(LEFT);
  Wait(TURN_3);
  r += 1;
}

else if(LIGHTLEFT <= 51 && LIGHTRIGHT >= 51 && LIGHTCENTER >= 51)
{
  SetPower(LEFT+RIGHT, MOTORPOWER_1);
  Off(LEFT+RIGHT);
  OnFwd(LEFT+RIGHT);
  Wait(TURN_CENTER);
  OnRev(LEFT);
  OnFwd(RIGHT);
  Wait(TURN_3);
  r -= 1;
}

else if(LIGHTLEFT >= 51 && LIGHTRIGHT >= 51 && LIGHTCENTER >= 51)
{
  SetPower(LEFT+RIGHT, MOTORPOWER_1);
  Off(LEFT+RIGHT);
  OnFwd(LEFT+RIGHT);
  Wait(CROSSING_CENTER);
  OnRev(RIGHT);
  OnFwd(LEFT);
  Wait(TURN_2);
  r -= 2;
}

else if(LIGHTLEFT <= 51 && LIGHTRIGHT >= 51 && LIGHTCENTER <= 51)
{
  SetPower(LEFT+RIGHT, MOTORPOWER_3);
  OnFwd(RIGHT);
  Float(LEFT);
  Wait(1);
}

else if(LIGHTLEFT >= 51 && LIGHTRIGHT <= 51 && LIGHTCENTER <= 51)
{
```

```
    SetPower(LEFT+RIGHT, MOTORPOWER_3);
    OnFwd(LEFT);
    Float(RIGHT);
    Wait(1);
  }

  else if(LIGHTLEFT >= 51 && LIGHTRIGHT >= 51 && LIGHTCENTER <= 51)
  {
    SetPower(LEFT+RIGHT, MOTORPOWER_1);
    OnFwd(LEFT+RIGHT);
    Wait(2);
  }
}

else if(end == 1)
{

  if(LIGHTCENTER >= 41 && LIGHTCENTER <= 42)
  {
    SetPower(LEFT+RIGHT, MOTORPOWER_2);
    Off(LEFT+RIGHT);
    OnFwd(LEFT+RIGHT);
    Wait(CROSSING_CENTER);

      if(a == 1)
      {
        OnRev(RIGHT);
        Wait(TURN_1);
      }

      else if(a == 2)
      {
        Wait(5);
      }

      else if(a == 3)
      {
        OnRev(LEFT);
        Wait(TURN_1);
      }
  }

  else if(LIGHTCENTER >= 35 && LIGHTCENTER <= 36)
  {
    Off(LEFT+RIGHT);
    PlayTone(220, 5);
    Wait(1000);
    end = 0;
  }

  else if(LIGHTCENTER >= 46 && LIGHTCENTER <= 47)
  {
    SetPower(LEFT+RIGHT, MOTORPOWER_2);
    Off(LEFT+RIGHT);
    OnFwd(LEFT+RIGHT);
    Wait(CROSSING_CENTER);

      if(b == 1)
      {
        OnRev(RIGHT);
        Wait(TURN_1);
      }
```

```
            else if(b == 2)
            {
              Wait(5);
            }

            else if(b == 3)
            {
              OnRev(LEFT);
              Wait(TURN_1);
            }
          }

          else if(LIGHTLEFT >= 51 && LIGHTRIGHT <= 51 && LIGHTCENTER >= 51)
          {
            SetPower(LEFT+RIGHT, MOTORPOWER_1);
            Off(LEFT+RIGHT);
            OnFwd(LEFT+RIGHT);
            Wait(TURN_CENTER);
            OnRev(RIGHT);
            OnFwd(LEFT);
            Wait(TURN_3);
          }

          else if(LIGHTLEFT <= 51 && LIGHTRIGHT >= 51 && LIGHTCENTER >= 51)
          {
            SetPower(LEFT+RIGHT, MOTORPOWER_1);
            Off(LEFT+RIGHT);
            OnFwd(LEFT+RIGHT);
            Wait(TURN_CENTER);
            OnRev(LEFT);
            OnFwd(RIGHT);
            Wait(TURN_3);
          }

          else if(LIGHTLEFT <= 51 && LIGHTRIGHT >= 51 && LIGHTCENTER <= 51)
          {
            SetPower(LEFT+RIGHT, MOTORPOWER_3);
            OnFwd(RIGHT);
            Float(LEFT);
            Wait(1);
          }

          else if(LIGHTLEFT >= 51 && LIGHTRIGHT <= 51 && LIGHTCENTER <= 51)
          {
            SetPower(LEFT+RIGHT, MOTORPOWER_3);
            OnFwd(LEFT);
            Float(RIGHT);
            Wait(1);
          }

          else if(LIGHTLEFT >= 51 && LIGHTRIGHT >= 51 && LIGHTCENTER <= 51)
          {
            SetPower(LEFT+RIGHT, MOTORPOWER_1);
            OnFwd(LEFT+RIGHT);
            Wait(2);
          }
        }
      }
    }
```

Appendix D. Test track